Article

A Reference Paper Collection System Using Web Scraping

Inzali Naing, Soe Thandar Aung, Khaing Hsu Wai and Nobuo Funabiki *

Department of Information and Communication Systems, Okayama University, Okayama 700-8530, Japan; psdy1zwf@s.okayama-u.ac.jp (I.N.); soethandar@s.okayama-u.ac.jp (S.T.A.); khainghsuwai@s.okayama-u.ac.jp (K.H.W.)
* Correspondence: funabiki@okayama-u.ac.jp

Abstract: Collecting reference papers from the Internet is one of the most important activities for progressing research and writing papers about their results. Unfortunately, the current process using *Google Scholar* may not be efficient, since a lot of paper files cannot be accessed directly by the user. Even if they are accessible, their effectiveness needs to be checked manually. In this paper, we propose a *reference paper collection system* using *web scraping* to automate paper collections from websites. This system can collect or monitor data from the Internet, which is considered as the environment, using *Selenium*, a popular *web scraping* software, as the sensor; this examines the similarity against the search target by comparing the keywords using the *Bert model*. The *Bert model* is a deep learning model for *natural language processing (NLP)* that can understand context by analyzing the relationships between words in a sentence bidirectionally. The *Python Flask* is adopted at the web application server, where *Angular* is used for data presentations. For the evaluation, we measured the performance, investigated the accuracy, and asked members of our laboratory to use the proposed method and provide their feedback. Their results confirm the method's effectiveness.

Keywords: web scraping; Google Scholar; data collection; Bert; Selenium; flask framework; Angular

check for updates

Citation: Naing, I.; Aung, S.T.; Wai, K.H.; Funabiki, N. A Reference Paper Collection System Using Web Scraping. *Electronics* **2024**, *13*, 2700. https://doi.org/10.3390/electronics13142700

Academic Editor: In Lee

Received: 29 April 2024
Revised: 21 June 2024
Accepted: 26 June 2024
Published: 10 July 2024

1. Introduction

Accessing research papers from the Internet is a crucial task for researchers. It helps them build upon existing knowledge and develop their own scholarly work. However, depending solely on a common platform like *Google Scholar* presents challenges. It can be found that, after searching, many papers are inaccessible due to pay barriers or access restrictions. Even when researchers can obtain the papers, they often need to spend a lot of time verifying their credibility and relevance manually. This manual checking process is a very time-consuming task and can slow down the research progress. Therefore, a better method is necessary to find and evaluate relevant research papers automatically, which would be beneficial for a lot of researchers by making their research more efficient and less time-consuming.

In searching for relevant reference papers using *Google Scholar* [1], researchers often encounter obstacles such as the difficulty of distinguishing between high-quality papers and low-quality ones, accessing full-text versions, and identifying the most relevant articles from a wide range of search results. Therefore, there is a growing need for a better search method to ensure the trustworthiness and relevance of papers. As researchers need to move through a large amount of information, simplifying access and verification processes becomes increasingly crucial for enhancing and speeding up their research.

In light of this challenge, the development of a system adopting an innovative technology, such as *web scraping* [2], may offer a promising solution. By utilizing *web scraping* technology, researchers can retrieve relevant papers using simple inputs, such as paper titles and keywords from their own papers. The result output of the system is expected to be a list of downloadable portable document format (PDF) file links with their titles indicating that they are relevant papers. Additionally, this system can validate the PDF links, checking the relevance and streamlining the validation process, in order to save researchers from the overwhelming burden of manual searches for relevant papers from the Internet. Moreover, to enhance user responsiveness, the system is implemented through the incorporation of *multi-threading* [3].

In this paper, we present a *reference paper collection system* using *web scraping* technology. It involves utilizing innovative technologies within the framework of interactive web services and *Docker* [4] for deployment. The system employs the *Bert model*, a *natural language processing (NLP)* technique from the *Hugging Face* library [5], to refine search results and enhance the accuracy and relevance in extracting references. *Selenium WebDriver* [6] plays a crucial role in controlling web pages on a browser and collecting data from them [7]. The system comprises *client-side (front end)* and *server-side (back end)* components. The entire architecture is maintained and managed as a repository using *Git* version control and is containerized via *Docker*. The front end, serving as the web user interface, is built using *Angular* [8,9]. The back end, driven by *Python Flask* [10], functions as the web application server. Furthermore, the adoption of the design architecture prioritizes generalizing the functionality, features, and common technology usage to optimize the running time, the memory utilization, and the performance of the central processing unit (CPU).

For the evaluation of the proposed system, we asked the members in our laboratory in Okayama University, Japan, to install the platform and to conduct paper searches related to their topics. Then, through comparisons with manual extractions after using *Google Scholar* directly, we confirmed the effectiveness and accuracy of the proposed platform in various research projects. By contrasting the two outputs, we assessed how well our automated approach aligned with human-generated results.

The rest of this paper is organized as follows. Section 2 introduces related works in the literature. Section 3 introduces the adopted software tools. Section 4 presents the implementation of the *reference paper collection system*. Section 5 evaluates the proposed system. Section 6 concludes this paper with a discussion and future works.

2. Related Work

In this section, we discuss papers related to *scholarly databases* and *web scraping for big data collection* in the literature.

2.1. Scholarly Databases

Google Scholar [11] is the most popular scholarly website that includes a wide range of research articles, theses, books, conference papers, and patents. The user interface of *Google Scholar* makes it easy for users to search and access scholarly content. The citation information can help researchers to evaluate the reputation and popularity of the paperwork. A user can initiate a new search by entering keywords into the search box. Articles can be filtered by author name and publication year in the advanced search setting. It will yield the result in the form of a comprehensive list of articles, which usually spans more than 10 pages.

Other scholarly databases like the Institute of Electrical and Electronics Engineers (*IEEE Xplore*) [12] and *Scopus* [13] primarily focus on technical and scientific papers and theses. Most of the papers are not open source, and the search interfaces of these databases may pose challenges for users in terms of finding and accessing detailed contents. In contrast to the user-friendly experience provided by *Google Scholar*, the navigation of search results on these platforms may require additional effort and familiarity with the specific databases. *Google Scholar* provides a more flexible search experience in comparison, allowing a user to retrieve a result with general keywords. On the other hand, the *Institute of Electrical and Electronics Engineers (IEEE Xplore)* interface requires specific and targeted keywords for locating the desired papers.

The wide range of search results on *Google Scholar* presents certain challenges. Navigating through suggested web pages to find relevant articles can be time-consuming. To ascertain an article's relevance, a user must open the link to access the PDF. Unfortunately, there are search results where PDF links or web pages are not available. Nevertheless, the convenience of obtaining search results with general keywords remains a noticeable advantage for *web scraping* in this research.

2.2. Web Scraping for Big Data Collection

Web scraping [14] is the automated information extraction process from websites. It is a technique for collecting large amounts of data on the Internet. Such data can be referred to as *Big Data* [15]. In the context of *Big Data* [16], *web scraping* revolves around navigating websites, systematically gathering a large volume of data, and converting the unstructured data commonly found in a *hyper text markup language (HTML)* format on a web page into the organized data suitable for storage and analysis [17].

Web scraping includes step-by-step operations such as data collection, extraction, transformation, and loading. *Data collection* and *data extraction* can be easily performed by *web scraping* tools and scripts. However, challenges arise in handling technical issues, such as web server maintenance, or when a web page is deleted after a certain period. Real-time *data collection* through *web scraping* has to include robust error-handling mechanisms that are able to respond to unexpected conditions. Additionally, before initiating *web scraping*, it is crucial to observe the privacy policies of the target website. Defining the specific data to be scraped in advance helps to minimize unnecessary data retrievals, thereby optimizing efficiency. The scraped data, often unstructured or semi-structured, requires subsequent cleaning and structuring in the data-cleaning phase.

Data transformation occurs after *data extraction*, converting the obtained data into various formats such as *JavaScript Object Notation (JSON)* (key–value pairs), *Extensible Markup Language (XML)*, or *Arrays*. These formatted datasets can then be stored in the file system or the database. Subsequently, the loaded data are integrated into a data warehouse, facilitating further research or addressing specific business needs.

After *data transformation*, loading data into a warehouse or storing data in a database or cloud is crucial for future use.

3. Adopted Tools

In this section, we introduce the software tools adopted in this study.

3.1. Google Scholar

Google Scholar is a search engine provided by *Google*. It offers free access to scholarly literature including papers, theses, and books. It indexes materials from various websites and allows users to search them by author name, title, or keyword. It provides citation metrics and features like personalized recommendations and alerts for new publications, serving as a valuable tool for researchers and academics.

3.2. Selenium

Selenium WebDriver is a popular web scraping tool [18]. It supports various browser–drivers, such as *ChromeDriver* for *Chrome*, *GeckoDriver* for *Firefox*, and *OperaDriver* for *Microsoft Edge*. To initiate *Selenium WebDriver*, the command script should be predefined. The script can be written in different programming languages, such as *Python*, *C#*, *Java*, *Ruby*, and *JavaScript*. Since *Selenium* [19] has the ability to control and navigate web pages through a browser, we used it to collect large amounts of data as a sensor environment.

In this paper, we made *Python* scripts [20] to function in *Selenium* as the sensor for collecting data from the Internet through *Google Scholar*. Upon receiving input from the user, this sensor program navigates to web pages to find downloadable PDFs, retrieves the data, and interacts with web page elements by clicking buttons, extracting information, completing forms, and waiting for responses.

3.3. BERT

BERT (Bidirectional Encoder Representations from Transformers) [5] is a transformer model developed by *Google*. Prior models such as *Word2Vec* [21], *GloVe,* [22] or *FastText* [23] are one-directional models, which can only read the input sequence from left to right and right to left sequentially. On the other hand, *BERT* has the ability to simultaneously read the entire context of words bidirectionally [24]. Self-attention mechanisms distinguish the importance of different words in the sentence. Due to these features, *BERT* has a more precise capability of understanding sentences without being biased to surrounding words [25].

The *sentence transformer* model [26] is a model that is pre-trained on a large dataset. It uses *BERT* as an *encoder*. The dataset used during training is collected from a large corpus, including *Wikipedia* [27], and by crawling data on the Internet. It calculates similarity scores according to the following procedure.

1. The *sentence transformer* model encodes each input text into the vector representation, which is also known as *embedding*, using a pre-trained transformer model such as *BERT* or the *Robustly Optimized BERT Approach (RoBERTa)*.
2. After generating *embedding* for the two texts, the *sentence transformer* model computes the similarity score using a metric like *Cosine similarity* or *Euclidean distance.*
3. The similarity score is based on the chosen similarity metric, with the higher score indicating a greater similarity between texts.

In this paper, we applied the *sentence transformer* model to compute similarity scores to extract the relevant papers from the results by *Google Scholar.*

3.4. Angular

Angular enables developers to create dynamic single-page web applications using *HTML*, *Cascading Style Sheets (CSS)*, and *TypeScript*. It is an open-source web application framework developed by *Google*. It utilizes the *model–view–controller (MVC)* architecture to effectively organize codes and separate concerns. *Angular* offers useful functions for routing, data binding, and dependency injections for facilitating the development of large-scale applications. Its *command line interface (CLI)* simplifies project creation, testing, and deployment, enhancing the developer's productivity. With robust features and strong community support, *Angular* is the preferred choice in many modern web application developments. As a front end framework, *Angular* provides an interactive user interface.

3.5. Python Flask

Flask is a lightweight web framework written in *Python* that adheres to the *WSGI (Web Server Gateway Interface)* specification. It enables seamless integration with various web servers. Despite its simplicity, *Flask* provides essential features such as routing, templating, and request handling for web applications. It also supports extensions for adding functionality, making it ideal for small–medium-sized web applications, *application programming interfaces (APIs)*, and prototypes. In this paper, *Python Flask* is used in the back end server to automate the *web scraping*, pass the keywords to the *sentence similarity* function, and provide responses to the front end.

4. Proposal of Reference Paper Collection System

In this section, we present the *reference paper collection system* by *web scraping*.

4.1. Software Architecture

The system is implemented as a web application for helping to facilitate academic paper searches and collect reference papers by researchers. It comprises two main components: the *user interface* of the *client side* and the *model part* of the *server side*. Together, these process *web scraping* and *data processing*. For the client side, *Angular* is utilized to implement the *user interface* for filling out the title and keywords for a new search on a browser. For the server architecture, *Python* is used in conjunction with *Flask*, which handles *web scraping* and *data processing* using *Selenium* and the *BERT* model, respectively. Figure 1 shows the overview of the system design.

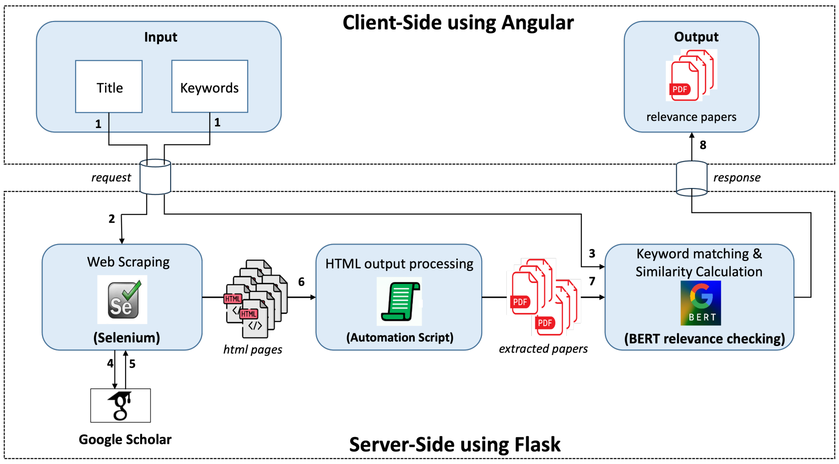


Figure 1. Overview of the system design.

4.2. Client-Side Implementation

For *client-side* implementation, *Angular* is adopted. Then, *Node.js* [28] version 12 is installed alongside *npm (Node Package Manager)*, which is crucial for managing dependencies [29]. For *Angular* developments, *TypeScript* [30] is used instead of *JavaScript* for creating scalable and easy-to-maintain applications. Additionally, *Material UI* [31], which is an open-source UI component library optimized for *Angular* applications, has been integrated. It ensures consistent user experiences across various platforms.

The implemented function is designed to simplify the process of searching for titles and keywords related to the user's input. The title may vary based on the user's choices. After the application is launched via the port *localhost:4200*, the user interface implementation logic works as shown in Algorithm 1. After the user clicks the submit button, the client side sends Hypertext Transfer Protocol *(HTTP)* requests along with the user-input data while showing the loading indicator. It then fetches the data from the server side. If the *server-side* response is successful and the web scraping procedure using *Selenium* and *BERT* is completed, then the user can view the pages of the resultant papers on the browser.

Algorithm 1 User interface implementation logic.

```
1:  function PAPERSEARCHFUNCTION(titleInput, keywordsInput)
2:      if $titleInput$ is empty or $keywordsInput$ is empty then
3:          display "Both fields are required."
4:          return
5:      end if
6:      display loadingIndicator
7:      $requestData \leftarrow$ { "title": $title$, "keywords": $keywords$ }
8:      $responseData \leftarrow$ FETCHPAPERS($requestData$)
9:      hide loadingIndicator
10:     if $responseData$ is empty then
11:         display "No relevant papers found."
12:     else
13:         for all $paper$ in $responseData$ do
14:             showData(paper)
15:         end for
16:     end if
17: end function
```

Connection between Server Side and Client Side

In our application architecture, a user interaction with the client side that is made on *Angular* and is accessible via *localhost:4200*, which initiates a sequence of events that involve communications with the *server side*; these are implemented with *Flask* and hosted at *localhost:5001*. When a user inputs data, such as *title* and *keywords*, and submits them, *Angular* generates the HTTP request containing these data. This request is directed towards the server's address *localhost:5001*, where the *Flask* application resides. Upon receiving the HTTP request, *Flask* processes the data within the request body. This typically involves extracting the submitted *title* and *keyword*. Subsequently, *Flask* proceeds to execute essential operations for *web scraping* and *data processing* based on the received data. Following this processing, *Flask* generates the response to be sent back to the client side. *Angular* then handles this response by updating the user interface as required, completing the interaction cycle between the *client-side* and *server-side* components.

4.3. Server-Side Implementation

For the *server side*, *Flask* handles the API requests from the *client side*. *Flask* provides *REST API* services using *Python* [32]. Upon receiving the request, typically via the *POST* method with additional parameters to the designated port *localhost:5001*, *Flask* initiates a series of procedures. Firstly, it uses *web scraping* techniques to add more information to the input data by obtaining extra details from different websites.

With the help of *automation scripts*, *Flask* handles the *HTML* documents to extract important information. *Flask* performs *keyword matching* and *similarity score calculations* to enhance the relevance and accuracy of the search results or recommendations. Then, *Flask* displays the selected papers on the *client-side* interface. We will explore each of these steps more comprehensively.

4.3.1. Web Scraping as Input Data Processing

Selenium and *Chrome Driver* are deployed as the *web scraping* environment in the system. In *Selenium version-3*, *JSON Wire Protocol* serves as the bridge between *Selenium* scripts and browser drivers, due to *Selenium version-3* lacking *World Wide Web Consortium (W3C)* standardization [33]. It encodes *Selenium* HTTP requests and sends them to browser drivers that are compliant with the *W3C protocol*. The *W3C protocol* decodes the response and relays the data back to the *JSON Wire Protocol* [34].

In this paper, we use the latest version *Selenium version-4*. It has the *World Wide Web Consortium (W3C)* standard as shown in Figure 2. *JSON Wire Protocol* is no longer needed to bridge the gap between the *Selenium* script and the browser driver [35]. *Selenium* can transfer the automation script directly to the driver via *W3C protocols* and execute the script in the browser. Regarding the *web scraping process* for input data, after retrieving the parameters such as the title and keywords from the *client side*, the automation script proceeds to search *Google Scholar* using the title. This step involves navigating to the *Google Scholar* website and initiating a search query with the provided title. The automation script interacts with the search result page, retrieves the *HTML* script from the browser, and forwards it to the scraping script for parsing.

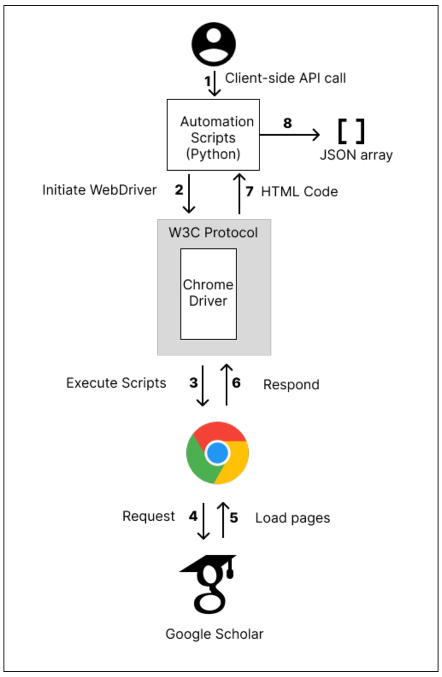
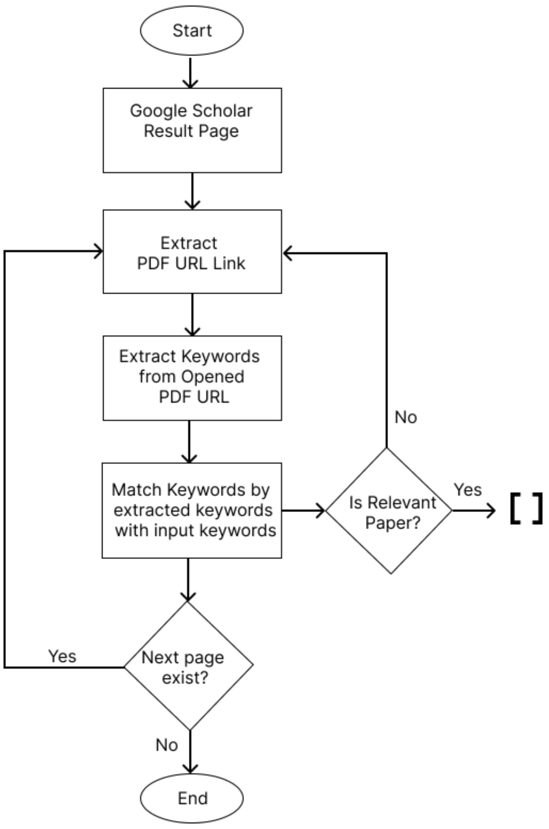


Figure 2. Web scraping with *Selenium version-4*.

4.3.2. HTML Output Processing with Automation Scripts

After obtaining the *HTML* output through *W3C protocol*, the data need to be scraped, focusing primarily on open-source reference papers. Typically, these papers are accessible in PDF format via online Uniform Resource Locators (URLs). To collect these URLs, we create the automation script. Figure 3 shows the flow chart of the *HTML* output processing.



Start
Google Scholar Result Page
Extract PDF URL Link
Extract Keywords from Opened PDF URL
Match Keywords by extracted keywords with input keywords
Is Relevant Paper?
Yes
[ ]
No
Next page exist?
Yes
No
End

Figure 3. Flow chart of *HTML* output process.

1. **Process of Automation Script:** After the *Google Scholar* results list loads, the *automation script* locates the URL link of the PDF using *XPath* [36]. The dynamic nature of locating *DOM (document object model)* elements on the website arises due to some tags lacking unique IDs or names. *XPath* is employed to pinpoint the PDF keywords among the DOM elements. Subsequently, the *regular expression* and the *Python* PDF reader library are utilized to extract keywords from the PDF's online URL. Once the keywords are obtained, the *sentence-transformer*-based *BERT* model is used to calculate the similarity score. Further details on data processing using the *sentence transformer* model will be provided in the next section. If a found paper is determined to be relevant, then key information is collected from *Google Scholar*, including the paper title, the citation count, and the link to the PDF's online URL according to Algorithm 2. Then, these data are transformed into a *JSON* array for the further visualization of the results.

Algorithm 2 Scraping data.

```
function ScrapingData(url, keywords)
    Navigate to url using ChromeDriver
    Initialize empty list paper_data
    for i ← 1 to 10 do
        while PDF links exist AND are valid do
            if CalculateSimilarity(paper.pdf_link, keywords) then
                paper.pdf_link ← driver.(By.XPATH, './/span[contains(text(), "[PDF]")]')
                paper.similarity ← CalculateSimilarity(paper.pdf_link, keywords)
                paper.title ← driver.get('title')
                paper.citation ← driver.get('citation')
                Add paper to paper_data
            end if
        end while
    end for
    return paper_data
     SSLError
    return FALSE
     HTTPError
    return FALSE
     RequestException
    return FALSE
    driver.close
end function
```

2. **Multi-Threading:** Sequential retrieval of each page can be time-consuming, since it requires waiting for all pages to be searched and processed. According to Algorithm 3 and Figure 4, the *multi-threading* mechanism [37] is employed to enhance the system response time. First, the total number of pages is calculated. Then, the web scraping process across these pages is distributed to different threads, which significantly reduces the processing time. However, increasing the number of threads also increases complexity. To ensure efficient system resource utilization, the available system resources are monitored.
3. **Error Handling Process:** Encountering dead links or invalid PDF URL links poses challenges. These links can lead to potential system crashes when *Python libraries* like *PyPDF2* [38] and *PyMuPDF* [39] fail to read PDF texts. To address this challenge, we have implemented *error handling mechanisms*, including the validation of the PDF link, page-not-found errors, and SSL (secure socket layer) errors. Moreover, repeatedly accessing a large number of PDF links can create challenging cases. Extracting the URL and opening the link several times can trigger the browser to identify the behavior as human activity. To overcome this problem, we utilize headless *Chrome*, which lacks

a *graphical user interface (GUI)*, consumes fewer system resources, and enables faster interactions. Additionally, we implemented the strategy to mimic human behaviors when interacting with the headless browser by introducing randomized time frames using waits and slower scrolling times, which can help avoid bot detection.

Algorithm 3 Multi-threading.

```
function MultiThreading(title, keywords)
    Initialize empty list urllist
    Construct base URL for Google Scholar search with the given title
    base_url ← 'https://scholar.google.com/scholar?q=' + ReplaceSpaces(title)
    Initialize ChromeDriver with base_url
    Count total result pages
    for page ← 1 to totalPages do
        url ← base_url + '&?page=' + page
        Append url to urlList
    end for
    Initialize empty list results
    Initialize threads or async tasks to scrape data from url
    for each taskUrl in urlList do
        data ← ScrapingData(taskUrl, keywords)
        result ← taskUrl.execute(data)
        Append result to results
    end for
    return results
end function
```

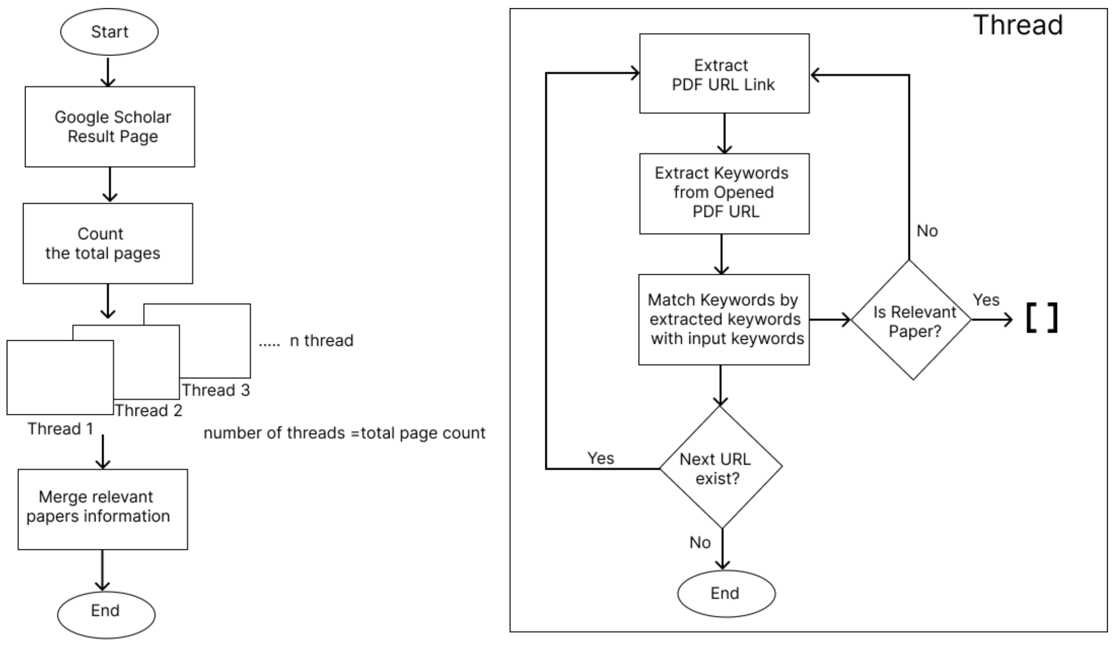

Start
Google Scholar Result Page
Count the total pages
Thread 1
Thread 2
Thread 3
..... n thread
number of threads =total page count
Merge relevant papers information
End
Thread
Extract PDF URL Link
Extract Keywords from Opened PDF URL
Match Keywords by extracted keywords with input keywords
Is Relevant Paper?
No
Yes
[ ]
Next URL exist?
Yes
No
End

Figure 4. Multi-threading for enhanced system response time in web scraping.

4.3.3. Keyword Matching and Similarity Score Calculation

To proceed to the *data processing* stage after the data extraction stage, one of the *natural language processing* techniques is applied to recommend the most relevant papers to the user. *BERT* with the *encoder-based sentence transformer* model is employed to find relevant papers for the user. *BERT* encodes the user's input keywords and the extracted keywords by *web scraping* into vector representations of the text.

The *sentence transformer* model computes the similarity score using the *cosine similarity*. To perform the process, the *Python sentence transformer* library is imported from

Hugging Face [40], which is a community providing open-source *large language models (LLMs)*. According to Algorithm 4, the *Python* program is made to pass two text arguments into the model: (1) the keywords extracted from the online PDF using the *Python* PDF reader and the regular expressions; (2) the user input text, separated by commas.

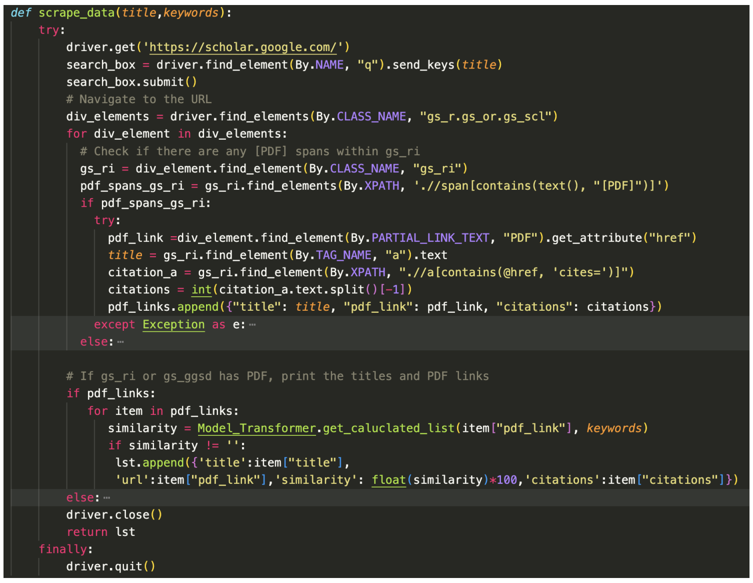
The *all-MiniLM-L12-v2*, which is one of the sentence transformer models [41], is known for its faster response time and smaller size, and is used to compute the similarity score between the user input keywords and the extracted papers' keywords. After the data processing stage, the server program returns the sorted list of the similarity scores to the *client-side* program. Further details on the visualization result outputs will be provided in the next section.

Algorithm 4 Calculate similarity.

```
 1: function CALCULATESIMILARITY(pdf_url, keywords)
 2:     if PDF is readable then
 3:         content ← READ(pdf_url)
 4:         terms ← ['Index Terms', 'keywords', 'Key Words']
 5:         matched_keywords ← REGEX_SEARCH(content, terms)
 6:         model ← SENTENCETRANSFORMER('sentence-transformers/all-MiniLM-L6-v2')
 7:         similarity ← MODEL.CALCULATE_SIMILARITY(keywords, matched_keywords)
 8:         return similarity
 9:     else
10:         return False
11:     end if
12: end function
```

4.4. Source Code Implementation for Back End Automation

As shown in Figure 5, the Python automation code is designed to search for academic papers on *Google Scholar* using specific keywords. It finds papers that have *PDF* versions available by handling multiple exceptions and collects information about these papers, such as their titles, links to the *PDFs*, and the number of times they have been cited by other papers. Afterward, the similarity is calculated using a *sentence-similarity transformer model*, which compares the provided keywords with those in the *PDFs* and returns a sorted list of academic papers.



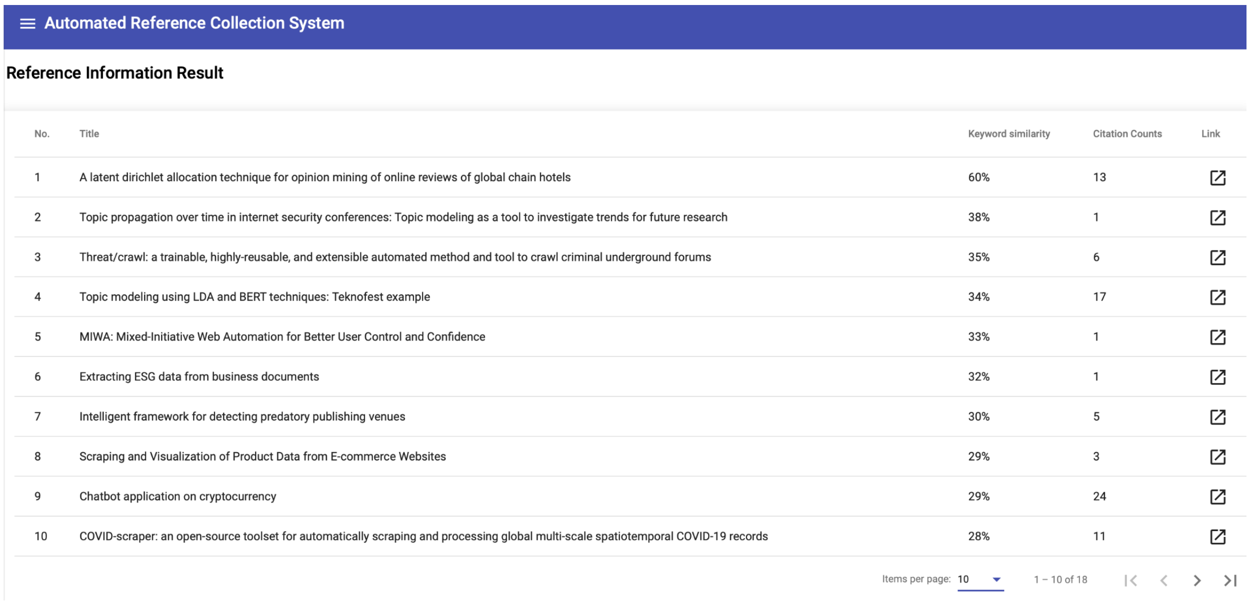
```python
def scrape_data(title,keywords):
    try:
        driver.get('https://scholar.google.com/')
        search_box = driver.find_element(By.NAME, "q").send_keys(title)
        search_box.submit()
        # Navigate to the URL
        div_elements = driver.find_elements(By.CLASS_NAME, "gs_r.gs_or.gs_scl")
        for div_element in div_elements:
          # Check if there are any [PDF] spans within gs_ri
          gs_ri = div_element.find_element(By.CLASS_NAME, "gs_ri")
          pdf_spans_gs_ri = gs_ri.find_elements(By.XPATH, './/span[contains(text(), "[PDF]")]')
          if pdf_spans_gs_ri:
            try:
              pdf_link =div_element.find_element(By.PARTIAL_LINK_TEXT, "PDF").get_attribute("href")
              title = gs_ri.find_element(By.TAG_NAME, "a").text
              citation_a = gs_ri.find_element(By.XPATH, ".//a[contains(@href, 'cites=')]")
              citations = int(citation_a.text.split()[-1])
              pdf_links.append({"title": title, "pdf_link": pdf_link, "citations": citations})
            except Exception as e:…
          else:…

        # If gs_ri or gs_ggsd has PDF, print the titles and PDF links
        if pdf_links:
           for item in pdf_links:
              similarity = Model_Transformer.get_caluclated_list(item["pdf_link"], keywords)
              if similarity != '':
               lst.append({'title':item["title"],
               'url':item["pdf_link"],'similarity': float(similarity)*100,'citations':item["citations"]})
        else:…
        driver.close()
        return lst
    finally:
        driver.quit()
```

Figure 5. Source code implementation of back end automation.

4.5. User Interface for Displaying Result Papers List

After completing the data processing stage, the back end server provides the reference papers in the *JSON* format. They are sorted according to the similarity score in descending order. Then, the *client side* receives the response and presents users with a visual representation of the sorted data. Figure 6 shows the user interface for displaying the resulting paper list.



Automated Reference Collection System

Reference Information Result

No.	Title	Keyword similarity	Citation Counts	Link
1	A latent dirichlet allocation technique for opinion mining of online reviews of global chain hotels	60%	13	
2	Topic propagation over time in internet security conferences: Topic modeling as a tool to investigate trends for future research	38%	1	
3	Threat/crawl: a trainable, highly-reusable, and extensible automated method and tool to crawl criminal underground forums	35%	6	
4	Topic modeling using LDA and BERT techniques: Teknofest example	34%	17	
5	MIWA: Mixed-Initiative Web Automation for Better User Control and Confidence	33%	1	
6	Extracting ESG data from business documents	32%	1	
7	Intelligent framework for detecting predatory publishing venues	30%	5	
8	Scraping and Visualization of Product Data from E-commerce Websites	29%	3	
9	Chatbot application on cryptocurrency	29%	24	
10	COVID-scraper: an open-source toolset for automatically scraping and processing global multi-scale spatiotemporal COVID-19 records	28%	11	

Items per page: 10 1 – 10 of 18

Figure 6. Interface for displaying resulting papers list.

5. Evaluations

In this section, we evaluate the proposed *reference paper collection system*. First, we evaluate the system performance using the *performance monitoring tool*. Second, we measure the accuracy of finding relevant reference papers in three cases. Last, we assess the effectiveness and usability through the questionnaire with the *system usability scale (SUS)* scores [42].

5.1. System Performance Evaluation

First, we evaluate the system performance in terms of both response time and memory usage in both the multi-threaded and conventional single-threaded cases [43]. We conduct these evaluations on a personal computer that is equipped with *Intel Core i7-8550U* CPU at 1.80 GHz with four cores and 16 GB memory, running *Windows 11*. To measure the performance metrics, we utilized the built-in tool *Performance Monitor* [44], provided by the *Windows* operating system. In the experiments, the system analyzed the first 10 pages of the search results from *Google Scholar*. Then, for the multi-thread case, 10 threads were used in parallel so that each thread handled a one-page search.

Table 1 shows the results of this test. The multi-thread execution was completed in 41 s with CPU usage of 10%, contrasting with the single-thread execution in 507 s with CPU usage of 2%. When compared with the single-thread execution, the multi-threaded approach resulted in an 8% increase in CPU usage and a reduction of 466 s in CPU time, with similar memory usage. With the use of multi-thread execution, the CPU time becomes acceptable for users.

Table 1. System performance results.

Parameter	Multi-Thread	Single-Thread
CPU time (s)	41s	507s
avg. CPU usage (%)	10%	2%
avg. memory usage (%)	46%	45%

5.2. Accuracy Evaluation

Second, we evaluate the system's accuracy in finding relevant reference papers for users by comparing it with *Google Scholar* and *Semantic Scholar* [45] across three cases with different research topics. *Semantic Scholar* is an AI-powered academic search engine that uses machine learning and natural language processing to help researchers find relevant academic papers across a wide range of disciplines. In each case, we used the same set of keywords for the proposed system, *Google Scholar*, and *Semantic Scholar*. The *system accuracy* represents the percentage of relevant accessible papers among all the papers suggested by the system and is calculated by Equation (1):

$$\text{accuracy} = \frac{\text{\# of relevant accessible papers}}{\text{total \# of suggested papers}} \times 100\% \tag{1}$$

where the relevant accessible paper represents a paper suggested by the system whose PDF file can be downloaded and whose contents are related to the research topic for search.

In the three cases, the following three factors are evaluated:

- **Relevance:** For *Google Scholar* and *Semantic Scholar*, we downloaded 100 papers from the first 10 pages of the result. For the proposed system, we downloaded all the papers suggested by the system. Then, the relevance of each paper was judged based on user preference, applicability to the target research, and similarity with the research.
- **Accessibility:** We compared the number of accessible and inaccessible papers within the first 10 pages of *Google Scholar*, *Semantic Scholar*, and the suggested papers by the proposed system.
- **Required time:** We compared the total time required to search the candidates, open the links to the PDF files, download them, and read them to determine relevance for the 100 papers from *Google Scholar* and *Semantic Scholar*, and all the suggested papers for the proposed system.

The results are presented in Tables 2–7 for *Case 1*–*Case 6*. These demonstrate that the proposed system achieves the highest accuracy in the shortest time. In *Case 6* and *Case 7*, the exact title of a published paper was used as the input. Then, *Semantic Scholar* returned only one result for *Case 6* and two results for *Case 7*, as shown in Table 7 and Table 8, respectively. Despite a very limited number of papers output by *Semantic Scholar*, the one paper in Table 7 was not relevant to this study and not accessible, whereas the two papers in Table 8 were both relevant and accessible.

Table 2. Accuracy of results in *Case 1*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	47	47%	58	58%	2 h 27 min
Semantic Scholar	100	58	58%	82	82%	2 h 18 min
Proposal	**33**	**25**	**75%**	**31**	**94%**	**45 min**

Table 3. Accuracy of results in *Case 2*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	41	41%	72	72%	2 h 18 min
Semantic Scholar	100	38	38%	63	63%	2 h 5 min
Proposal	**51**	**31**	**61%**	**51**	**100%**	**49 min**

Table 4. Accuracy of results in *Case 3*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	33	33%	72	72%	2 h 12 min
Semantic Scholar	100	22	22%	47	47%	1 h 58 min
Proposal	**37**	**16**	**43%**	**34**	**91%**	**46 min**

Table 5. Accuracy of results in *Case 4*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	28	28%	71	71%	2 h 20 min
Semantic Scholar	100	20	20%	49	49%	2 h 4 min
Proposal	**58**	**29**	**50%**	**52**	**89%**	**57 min**

Table 6. Accuracy of results in *Case 5*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	23	23%	63	63%	2 h 31 min
Semantic Scholar	100	11	11%	42	42%	2 h 24 min
Proposal	**25**	**11**	**44%**	**22**	**88%**	**30 min**

Table 7. Accuracy of results in *Case 6*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	28	28%	60	60%	2 h 42 min
Semantic Scholar	1	0	0%	0	0%	10 s
Proposal	**46**	**21**	**46%**	**37**	**80%**	**41 min**

Table 8. Accuracy of results in *Case 7*.

System	# of Papers	Relevance		Accessibility		Time
		# of Papers	Accuracy	# of Papers	Accuracy	
Google Scholar	100	26	26%	63	63%	1 h 50 min
Semantic Scholar	2	2	100%	2	100%	1 min
Proposal	**28**	**15**	**54%**	**25**	**89%**	**42 min**

5.2.1. Case 1

For *Case 1*, the title for search was "Web scraping using Selenium" and the keywords were "Selenium", "ChromeDriver", "Python", and "data collection". Table 2 shows the results.

5.2.2. Case 2

For *Case 2*, the title for search was "Sentence transformer and sentence similarity using Bert model" and the keywords were "embedding", "similarity", "attention layer", "Bert", and "NLP". Table 3 shows the results.

5.2.3. Case 3

For *Case 3*, the title for search was "Automated image-based UI testing for mobile applications" and the keywords were "user interface", "testing", "Image", "CSS", and "mobile platforms". Table 4 shows the results.

5.2.4. Case 4

For *Case 4*, the title for search was "Indoor navigation system" and the keywords were "indoor navigation", "Unity", "QR code", and "smartphone". Table 5 shows the results.

5.2.5. Case 5

For *Case 5*, the title for search was "Docker image generation" and the keywords was "Docker". Table 6 shows the results.

5.2.6. Case 6

For *Case 6*, the title for search was "A Study of Integrated Server Platform for IoT Application Systems" and the keywords were "IoT platform", "cloud", "communication", "smart cities", and "artificial intelligence". Table 7 shows the results.

5.2.7. Case 7

For *Case 7*, the title for search was "A proposal of code modification problem for self-study of web client programming using JavaScript" and the keywords were "Web client programming", "JavaScript", "HTML", "CSS", "code modification", "coding reading", and "self-study". Table 8 shows the results.

5.3. System Usability Evaluation

Finally, we investigated the system usability of the proposed system through a questionnaire with the *system usability scale (SUS)* scores. We asked 10 graduate students in our laboratory to install the proposed system on their personal computers (PCs) using *Docker*, and to use the system by inputting the title and keywords that are related to their current research topics. After that, we asked them to answer 10 questions to collect their feedback on the proposal.

Table 9 shows the 10 questions in this questionnaire and Table 10 indicates the corresponding answers from the 10 students. The average *SUS score* among them falls between 68 and 80.3, suggesting a "Good" level of usability based on the calculated SUS score interpretation in Table 11 [46].

Table 9. SUS Questions in questionnaire.

#	Question
1.	The search input interface is easy to use.
2.	The system responds too slowly.
3.	The output result list is easy to understand.
4.	To obtain a good result, I had to attempt the input several times.
5.	The output result papers are downloadable as PDF.
6.	Manual searching is more useful than this system.
7.	The result papers are relevant.
8.	I require technical support to set up the system on my PC.
9.	I would like to recommend this system to other people.
10.	I experienced bugs when using the system.

Table 10. Answers for SUS questionnaire and SUS scores.

User	Answers for SUS Questions										Overall SUS Scores
	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	
1	5	4	5	3	5	3	4	1	3	1	75
2	5	2	5	3	5	1	4	1	5	1	90
3	5	2	3	2	3	3	4	1	3	4	65
4	4	1	5	2	4	2	4	2	4	2	80
5	5	4	5	3	5	3	3	2	4	4	65
6	2	5	4	3	5	4	4	5	4	4	45
7	4	4	4	4	4	2	4	2	4	1	67.5
8	4	2	4	2	4	2	5	2	4	2	77.5
9	5	5	2	3	1	4	2	3	3	5	37.5
10	5	1	4	2	4	1	5	3	5	3	82.5
									Average SUS Score		68.5

Table 11. SUS score interpretation.

SUS Score	Grade	Adjective Rating
>80.3	A	Excellent
68–80.3	B	Good
68	C	Okay
51–68	D	Poor
<51	F	Awful

5.4. Feedback from Students

Unfortunately, 2 students out of 10 answered F grades in Table 11. Both students used PCs with Macintosh Operating System (MAC OS) with M1 (silicon chip), and encountered installation errors. The other students used PCs with (intel chip), Linux, or Windows and experienced no installation errors. To address this issue, we updated the installation manual and provided the option to use another PC with Windows.

Some students suggested allowing an input of keywords only, rather than requiring both keywords and a title. Some students suggested adding a publication date filter to find only papers within certain publication dates. Most students rated the proposed system as excellent or good and replied that it is more useful and accurate than manual paper collection from *Google Scholar* results.

6. Discussion and Conclusions

This paper presented a *reference paper collection system* using *web scraping* as a sensor to collect relevant papers from websites in the Internet environment. This system adopts *Selenium* as the *web scraping* tool and examines the similarity against the search target by comparing keywords using the *BERT model*. *Python Flask* is adopted as the web application server and *Angular* is used for the web interfaces. For the evaluation, we measured performance, investigated the accuracy of results, and asked members in our laboratory to use the proposed system and return their feedback. Their results confirmed the effectiveness of the system.

Our *web scraping* approach not only addresses the time-consuming task of manually collecting data but also provides accessible *PDF* file links. However, it is important to review and comply with the terms of service of the websites being scraped. Generally, *web scraping* is legal in Japan if it accesses publicly available information. However, scraping restricted or authorized content without permission is illegal. In this proposed system, we sought only open-access *PDFs* from academic websites using *Google Scholar*.

Although our proposed system provides a powerful method for collecting relevant academic papers, we acknowledge certain limitations related to *web scraping*. Unfortunately, the terms of service of *Google Scholar* do not allow automated scraping, even for accessing open-access *PDFs* from other websites using its service.

To address these challenges, we will integrate *APIs* from legitimate sources like *Unpaywall*, *CORE*, *Semantic Scholar*, and *CrossRef*. These *APIs* offer structured and lawful access to open-access research articles. Furthermore, transitioning to API-based data retrieval can enhance the system's reliability and reduce the risk of *Internet Protocol (IP)* banning during *web scraping*. Moreover, we will enhance the accuracy of the system by incorporating a publication date filter and *summarization model*, and validate the applicability of the approach for various research topics. Moreover, we will consider implementing the system on a cloud-based platform to improve accessibility for users.

Author Contributions: Conceptualization, I.N. and N.F.; methodology, I.N.; software, I.N.; formal analysis, I.N.; investigation, S.T.A. and K.H.W.; resources, S.T.A. and K.H.W.; visualization, I.N.; writing—original draft preparation, I.N.; writing—review and editing, I.N., S.T.A. and N.F; supervision, N.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Shultz, M. Comparing test searches in PubMed and Google Scholar. *J. Med. Libr. Assoc.* **2007**, *95*, 442–445. [CrossRef]
2. Han, S.; Anderson, C.K. Web scraping for hospitality research: Overview, opportunities, and implications. *Cornell Hosp. Q.* **2021**, *62*, 89–104. [CrossRef]
3. Naing, I.; Funabiki, N.; Wai, K.H.; Aung, S.T. A design of automatic reference paper collection system using Selenium and Bert Model. In Proceedings of the IEEE 12th Global Conference on Consumer Electronics (GCCE), Nara, Japan, 10–13 October 2023; pp. 267–268.
4. Docker. Available online: https://docs.docker.com/ (accessed on 24 April 2024).
5. Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 NAACL-HLT, Minneapolis, MN, USA, 2–7 June 2019; pp. 4171–4186.
6. Gojare, S.; Joshi, R.; Gaigaware, D. Analysis and design of selenium WebDriver automation testing framework. *Procedia Comput. Sci.* **2015**, *50*, 341–346. [CrossRef]
7. Lotfi, C.; Srinivasan, S.; Ertz, M.; Latrous, I. Web scraping techniques and applications: A Literature review. In Proceedings of the SCRS Conference on Intelligent Systems, SCRS, Delhi, India, 25 April 2022; pp. 381–394. [CrossRef]
8. Angular. Available online: https://angular.io/ (accessed on 24 April 2024).

9. Cincovic, J.; Delcev, S.; Draskovic, D. Architecture of web applications based on Angular framework: A case study. In Proceedings of the ICIST, Durham University, Durham, UK, 10–12 July 2019; pp. 254–259. Available online: https://api.semanticscholar.org/CorpusID:222459277 (accessed on 24 April 2024).
10. Vyshnavi, V.R.; Malik, A. Efficient way of web development using Python and Flask. *Int. J. Recent Res. Asp.* **2019**, *6*, 16–19.
11. Google Scholar. Available online: https://ja.wikipedia.org/wiki/Google_Scholar (accessed on 24 April 2024).
12. Wilde, M. IEEE Xplore digital library. *Charl. Adv.* **2016**, *17*, 24–30. [CrossRef]
13. Krauskopf, E. An analysis of discontinued journals by Scopus. *Scientometrics* **2018**, *116*, 1805–1815. [CrossRef]
14. Glez-Peña, D.; Lourenço, A.; López-Fernández, H.; Reboiro-Jato, M.; Fdez-Riverola, F. Web scraping technologies in an API world. *Brief. Bioinform.* **2014**, *15*, 788–797. [CrossRef] [PubMed]
15. Djedouboum, A.C.; Abba Ari, A.A.; Gueroui, A.M.; Mohamadou, A.; Aliouat, Z. Big data collection in large-scale wireless sensor networks. *Sensors* **2018**, *18*, 4474. [CrossRef] [PubMed]
16. Snell, J.; Menaldo, N. Web scraping in an era of big data 2.0. *Bloomberg Law News*, June 2016.
17. Landers, R.N.; Brusso, R.C.; Cavanaugh, K.J.; Collmus, A.B. A primer on theory-driven web scraping: Automatic extraction of big data from the Internet for use in psychological research. *Psych. Meth.* **2016**, *21*, 475–492. [CrossRef]
18. Wendt, H.; Henriksson, M. Building a Selenium-Based Data Collection Tool. Bachelor's Thesis, 16 ECTS, Information Technology, Linköping University, Linköping, Sweden, May 2020.
19. Selenium WebDriver. Available online: https://www.selenium.dev/documentation/webdriver/ (accessed on 24 April 2024).
20. Mitchell, R. *Web Scraping with Python: Collecting More Data from the Modern Web*, 2nd ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2018; pp. 1–308.
21. Church, K.W. Word2Vec. *Natur. Lang. Engin.* **2017**, *23*, 155–162. [CrossRef]
22. Pennington, J.; Socher, R.; D-Manning, C. GloVe: Global Vectors for Word Representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543.
23. Yao, T.; Zhai, Z.; Gao, B. Text Classification Model Based on fastText. In Proceedings of the 2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS), Dalian, China, 20–22 March 2020; pp. 154–157. [CrossRef]
24. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Transformers: State-of-the-art natural language processing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Online, 16–20 November 2020; pp. 38–45.
25. Sun, X.; Meng, Y.; Ao, X.; Wu, F.; Zhang, T.; Li, J.; Fan, C. Sentence similarity based on contexts. *Trans. Assoc. Comput. Ling.* **2022**, *10*, 573–588. [CrossRef]
26. Reimers, N.; Gurevych, I. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, Hong Kong, China, 3–7 November 2019; pp. 3982–3992. Available online: https://aclanthology.org/D19-1410.pdf (accessed on 24 April 2024).
27. Wikipedia. Available online: https://www.wikipedia.org/ (accessed on 24 April 2024).
28. Node.js. Available online: https://nodejs.org/en (accessed on 24 April 2024).
29. NPM. Available online: https://docs.npmjs.com/ (accessed on 24 April 2024).
30. TypeScript. Available online: https://www.typescriptlang.org/ (accessed on 24 April 2024).
31. Material UI. Available online: https://material.angular.io/ (accessed on 24 April 2024).
32. Flask. Available online: https://flask.palletsprojects.com/en/3.0.x/.v (accessed on 24 April 2024).
33. Dahl, D. The W3C multimodal architecture and interfaces standard. *J. Multi. User Inter.* **2013**, *7*, 171–182. [CrossRef]
34. Gundecha, U.; Avasarala, S. *Selenium WebDriver 3 Practical Guide: End-to-End Automation Testing for Web and Mobile Browsers with Selenium WebDriver*, 2nd ed.; Packt Publishing: Birmingham, UK, 2018; pp. 1–280.
35. Viotti, J.C.; Kinderkhedia, M. A Survey of JSON-Compatible Binary Serialization Specifications. *arXiv* **2022**, arXiv:2201.02089. Available online: https://arxiv.org/abs/2201.02089 (accessed on 24 April 2024).
36. XPath. Available online: https://developer.mozilla.org/ja/docs/Web/XPath (accessed on 24 April 2024).
37. Mustofa, K.; Fajar, S.P. Selenium-based multithreading functional testing. *Indones. J. Comput. Cybern. Syst. (IJCCS)* **2018**, *12*, 63–72. [CrossRef]
38. PyPDF2. Available online: https://pypi.org/project/PyPDF2/ (accessed on 24 April 2024).
39. PyMuPDF. Available online: https://pypi.org/project/PyMuPDF/ (accessed on 24 April 2024).
40. Hugging Face. Available online: https://huggingface.co/ (accessed on 24 April 2024).
41. all-MiniLM-L12-v2—Sentence Transformer Model. Available online: https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2 (accessed on 24 April 2024).
42. Measuring and Interpreting System Usability Scale—SUS. Available online: https://uiuxtrend.com/measuring-system-usability-scale-sus/ (accessed on 24 April 2024).
43. Zefeng, Q.; Umapathy, P.; Zhang, Q.; Song, G.; Zhu, T. Map-reduce for multiprocessing large data and multi-threading for data scraping, Mathematics—Numerical Analysis. *arXiv* **2023**, arXiv:2312.15158.
44. Performance Monitor. Available online: https://en.wikipedia.org/wiki/Performance_Monitor (accessed on 24 April 2024).

45. Kinney, R.; Anastasiades, C.; Authur, R.; Beltagy, I.; Bragg, J.; Buraczynski, A.; Cachola, I.; Candra, S.; Chandrasekhar, Y.; Cohan, A.; et al. The Semantic Scholar Open Data Platform. *arXiv* **2023**, arXiv:2301.10140. [CrossRef]
46. System Usability Scale—SUS. Available online: https://credoagency.co.uk/usability-in-cro-the-system-usability-scale-sus/ (accessed on 24 April 2024).