A
Capstone Project Report

on

Web-Sight: Automatic Web Scraping Tool

Submitted to

BHILAI INSTITUTE OF TECHNOLOGY, DURG

an Autonomous Institute

Affiliated to

CHHATTISGARH SWAMI VIVEKANAND TECHNICAL UNIVERSITY

BHILAI

of

Bachelor of Technology

in

Computer Science and Engineering

by

Rohit Kushwaha, 8th sem, 300102220127

Yaman Kumar Sahu, 8th sem, 300102220092

Vansh Dharmani, 8th sem, 300102220086

Under the Guidance of
Jyoti Gupta
Assistant Professor

Department of Computer Science and Engineering
Bhilai Institute of Technology,
(An Autonomous Institute)
Bhilai House, GE Road, Durg, Chhattisgarh 491001

Session: 2023 – 2024

DECLARATION BY THE CANDIDATE(s)

I/we the undersigned solemnly declare that the report of the project work entitled *Web-Sight: Automatic Web Scraping Tool*, is based on my own work carried out during the course of my study under the supervision of *Prof. Jyoti Gupta*.

I assert that the statements made and conclusions drawn are an outcome of the project work. I further declare that to the best of my knowledge and belief that the report does not contain any part of any work which has been submitted for the award of any other degree/diploma/certificate in this University/ any other University of India or any other country.

_____
(Signature of Student)
Name of the Student(s): Rohit Kushwaha
Roll No(s).: 30010220127
Enrollment No(s).: BK2553

_____
(Signature of Student)
Name of the Student(s): Yaman Kumar Sahu
Roll No(s).: 30010220092
Enrollment No(s).: BK4269

_____
(Signature of Student)
Name of the Student(s): Vansh Dharmani
Roll No(s).: 30010220086
Enrollment No(s).: BK4263

APPROVAL CERTIFICATE

This is to Certify that the report of the project submitted is an outcome of the project work entitled Web-Sight: Automatic Web Scraping Tool carried out by **Rohit Kushwaha, bearing Roll No 300102220127, Enrollment No BK2553; Yaman Kumar Sahu, bearing Roll No 300102220092, Enrollment No BK4269; Vansh Dharmani, bearing Roll No 300102220086, Enrollment No BK4263.**

Under my guidance and supervision in partial fulfillment of Bachelor of Technology in Computer Science from Bhilai Institute of Technology, Durg, an autonomous institute affiliated to Chhattisgarh Swami Vivekanand Technical University, Bhilai (C.G).

To the best of my knowledge and belief the project

i) Embodies the work of the candidate himself / herself,

ii) Has duly been completed,

iii) Fulfills the requirement of the Ordinance relating to the B. Tech. degree of the University,

iv) Is up to the desired standard for the purpose of which is submitted.

_____

(Signature of the Supervisor)

Mrs. Jyoti Gupta
Assistant Professor
Computer Science & Engineering.

The Project work as mentioned above is hereby being recommended and forwarded for examination and evaluation.

Dr. (Mrs,) Sunita Soni
Head of the Department
Computer Science & Engineering

CERTIFICATE BY THE EXAMINERS

This is to Certify that the project the entitled

"**Web-Sight: Automatic Web Scraping Tool",**

Submitted by

Rohit Kushwaha	**Enrollment No: BK2553**	**Roll No:300102220127**
Yaman Kumar Sahu	**Enrollment No: BK4269**	**Roll No:300102220092**
Vansh Dharmani	**Enrollment No: BK4263**	**Roll No:300102220086**

Have been examined by the undersigned as a part of the examination for the award of Bachelor of Technology degree in Computer Science and Engineering from Bhilai Institute of Technology, Durg, an autonomous institute affiliated to Chhattisgarh Swami Vivekanand Technical University, Bhilai (C.G)

(Internal Examiner) **(External Examiner)**

Name: **Name:**

Date: **Date:**

ACKNOWLEDGEMENT

I have great pleasure in the submission of this project report entitled **Web-Sight : Automatic Web Scraping Tool** in partial fulfillment the degree of Bachelor of Technology. While submitting this project report, I take this opportunity to thank those directly or indirectly related to project work.

I would like to thank my supervisor **Prof. Jyoti Gupta** who has provided the opportunity and organizing project for me. Without his active co-operation and guidance, it would have become very difficult to complete task in time.

I would like to express sincere thanks and gratitude to Dr. Arun Arora, **Principal of the Institution**, Dr. (Mrs.) Sunita Soni, **Head of the Department** Computer Science & Engineering for their encouragement and cordial support.

While Submission of the project, I also like to thanks to Prof. Shiv Dutta Mishra, and Prof. Vibhore Jain **Project Coordinator,** faculties and all the staff of department of Computer Science & Engineering, **Bhilai Institute of Technology, Durg** for their continuous help and guidance throughout the course of project.

Acknowledgement is due to our parents, family members, friends and all those persons who have helped us directly or indirectly in the successful completion of the project work.

Name of the Student(s):Rohit Kushwaha
Roll No(s):300102220127
Enrollment(s): BK2553

Name of the Student(s):Yaman Kumar Sahu
Roll No(s): 300102220092
Enrollment(s): BK4269

Name of the Student(s):Vansh Dharmani
Roll No(s): 300102220086
Enrollment(s): BK4263

ABSTRACT

This project aims to develop an AI-powered web scraping service that enables users to extract data from websites in a fast and efficient manner. The service utilizes a combination of web scraping techniques, natural language processing, and machine learning algorithms to automatically identify and extract relevant data from web pages. The service provides a user-friendly interface that allows users to specify the URL and the data they want to extract, and then generates a structured dataset that can be easily analyzed and visualized. The service also includes features such as data validation, error handling, and data formatting to ensure the accuracy and consistency of the extracted data. The project is implemented using a combination of Python, JavaScript, and HTML/CSS, and is designed to be scalable, reliable, and secure. The service has the potential to revolutionize the way businesses and researchers collect and analyze data from the web, and has numerous applications in fields such as market research, competitive intelligence, and data journalism.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER: 1
INTRODUCTION

Chapter 1

INTRODUCTION

1.1 Introduction

1.1.1 What is Web Scraping?

Web Scraping is an automatic method to obtain large amounts of data from websites. Most of this data is unstructured data in an HTML format which is then converted into structured data in a spreadsheet or a database so that it can be used in various applications. There are many different ways to perform web scraping to obtain data from websites. These include using online services, particular API's or even creating your code for web scraping from scratch. Many large websites, like Google, Twitter, Facebook, StackOverflow, etc. have API's that allow you to access their data in a structured format. This is the best option, but there are other sites that don't allow users to access large amounts of data in a structured form or they are simply not that technologically advanced. In that situation, it's best to use Web Scraping to scrape the website for data.

Web scraping requires two parts, namely the crawler and the scraper**.** The crawler is an artificial intelligence algorithm that browses the web to search for the particular data required by following the links across the internet. The scraper, on the other hand, is a specific tool created to extract data from the website. The design of the scraper can vary greatly according to the complexity and scope of the project so that it can quickly and accurately extract the data.

1.1.2 What is AI Web Scraping?

AI web scraping refers to the automated process of extracting data from websites using a few AI based methods. Unlike traditional web scraping which relies on pre-defined selectors that isolate data you want to extract, AI web scraping employs self-adjusting algorithms capable of handling dynamic websites. This approach addresses the limitations associated with manual or purely code-based scraping techniques - in short it's more effective. AI web scraping tools are designed to navigate through web pages, identify and extract relevant data, and adapt to changes in website layouts without human intervention.

"Our AI-based web scraping project utilizes cutting-edge technology to automatically extract valuable data from various online sources. Leveraging advanced machine learning algorithms and natural language processing techniques, our system can

intelligently navigate through web pages, identify relevant information, and extract it with precision and efficiency.

With the ability to scrape data from a diverse range of websites, including e-commerce platforms, news sites, social media platforms, and more, our project opens up a wealth of opportunities for businesses and researchers alike. Whether it's monitoring market trends, tracking competitor activity, or gathering insights for decision-making, our AI-powered solution delivers actionable data at scale.

By automating the scraping process and leveraging AI for data analysis, we enable users to unlock valuable insights faster and more effectively than ever before. Our project prioritizes data accuracy, privacy, and ethical scraping practices, ensuring compliance with regulations and maintaining trust with users.

With a focus on innovation and continuous improvement, we're committed to advancing our project with new features, enhancements, and optimizations. Join us on the cutting edge of web scraping technology and harness the power of AI to drive your data-driven initiatives forward."

1.2 Project Overview:

Our AI-based web scraping project revolutionizes the way data is extracted from the vast landscape of the internet. By combining artificial intelligence, machine learning, and sophisticated algorithms, we've developed a robust solution that automates the process of gathering data from diverse online sources.

1.2.1 Key Features:

1. Advanced Data Extraction: Our system employs state-of-the-art techniques to intelligently navigate through web pages and extract structured data. From product information on e-commerce websites to news articles and social media posts, our AI can identify and retrieve relevant data with high accuracy.

2. Dynamic Content Handling: With the ability to handle dynamic web content, including JavaScript-rendered pages, our project ensures comprehensive data extraction from modern websites. This capability enables us to capture real-time updates and changes, providing users with the most current information available.

3. Customizable Scraping Rules: Users can define custom scraping rules and configurations to tailor the extraction process to their specific requirements. Whether it's specifying target elements, setting data extraction frequency, or implementing data filters, our project offers flexibility and customization options to suit diverse use cases.

4. Scalability and Performance: Our solution is designed for scalability, capable of handling large-scale scraping tasks efficiently. By leveraging distributed computing and parallel processing techniques, we ensure optimal performance even when dealing with massive datasets and complex web structures.

5. Data Quality Assurance: Quality control mechanisms are integrated into the scraping pipeline to ensure the accuracy and integrity of the extracted data. Through validation checks, error handling, and anomaly detection algorithms, we maintain high standards of data quality and reliability.

1.3 Background and Problem Motivation:

1.3.1 Background:

In today's data-driven world, accessing and analyzing vast amounts of data from the web is crucial for various applications across industries. However, manually collecting data from websites can be time-consuming, labor-intensive, and error-prone. Web scraping, the automated process of extracting data from websites, addresses this challenge by enabling the efficient and systematic retrieval of structured information from the web.

1.3.2 Problem Motivation:

Despite its potential benefits, traditional web scraping methods face several limitations and challenges:

1. Scale and Efficiency: Manually maintaining and updating web scraping scripts for multiple websites and sources can be cumbersome, especially as the volume of data and complexity of web pages increase.

2. Dynamic Content: Websites often use dynamic content loading techniques such as AJAX, JavaScript, and single-page applications (SPAs), making it challenging to extract data reliably using conventional scraping techniques.

3. Data Quality and Integrity: Ensuring the accuracy, consistency, and integrity of scraped data can be difficult due to variations in website structures, data formats, and evolving web page layouts.

4. Legal and Ethical Concerns: Web scraping activities may raise legal and ethical concerns related to data privacy, copyright infringement, and compliance with website terms of service and usage policies.

5. Resource Intensiveness: Traditional web scraping methods may require significant computational resources, bandwidth, and network overhead, particularly when scraping large volumes of data or dealing with complex website structures.

1.4 Project Objectives:

The primary objectives of this AI web scraping project are as follows:

1. Automation: Develop and deploy AI-powered algorithms and techniques to automate the web scraping process, enabling efficient and scalable data extraction from diverse sources.

2. Accuracy and Robustness: Enhance the accuracy, robustness, and adaptability of web scraping methods to handle dynamic content, complex web page structures, and changes in website layouts.

3. Data Quality Assurance: Implement mechanisms for data validation, cleansing, and quality assurance to ensure the reliability and integrity of scraped data for downstream analysis and decision-making.

4. Compliance and Governance: Incorporate legal and ethical considerations into the web scraping workflow, including compliance with data protection regulations, respect for website terms of service, and adherence to ethical guidelines for responsible data usage.

5. Performance Optimization: Optimize the performance, efficiency, and resource utilization of web scraping algorithms and systems to minimize latency, maximize throughput, and reduce operational costs.

CHAPTER: 2
LITERATURE REVIEW

Chapter 2

LITERATURE REVIEW

2.1 Literature Review

The origin of the World Wide Web dates back to 1989, when British scientist Tim Berners-Lee conceived a platform for global information exchange among scientists. This groundbreaking creation introduced basic elements that form the foundation of modern web scraping tools: URLs for website designation, embedded hyperlinks for navigation, and web pages containing diverse data types.

In 1991, Berners-Lee created the first web browser, allowing access to the Web through HTTP pages hosted on servers, thereby facilitating interaction with online content.

Around the same time in 1993, Matthew Gray at MIT introduced the concept of web crawling with the creation of the World Wide Web Wanderer. This Perl-based web crawler aimed to measure the size of the web and contributed to the development of web indexing, as exemplified by the creation of the Wandex index.

Simultaneously in 1993, JumpStation emerged as the first crawler-based web search engine, laying the groundwork for subsequent search giants like Google, Bing, and Yahoo. With millions of web pages indexed, the internet transformed into a vast repository of accessible data.

Fast forward to 2004, the introduction of BeautifulSoup, an HTML parser library written in Python, revolutionized web scraping. This tool simplified site structure comprehension and content parsing, significantly reducing the workload for programmers.

As the internet evolved into an extensive information hub accessible to anyone with an internet connection, web scraping gained popularity as a means of extracting valuable data. Initially, websites didn't restrict data downloading, but as this practice grew, manual copy-pasting became impractical. Thus, alternative methods for data extraction became necessary.

Web scraping, the process of extracting data from websites, encompasses various techniques. Among these methods, traditional copy-and-paste stands out as a straightforward yet labor-intensive approach. This manual technique involves selecting data from a webpage and transferring it to another location for analysis. While effective for small-scale tasks, it becomes impractical and tedious when dealing with large datasets. Sirisuriya (2015) points out that this method can be error-prone and unpleasant due to its time-consuming nature.[2]

Table 2.1 Comparison of Web Scraping Techniques

YEAR	AUTHOR	PURPOSE	TECHNIQUES
2015	Sirisuriya	Discussing web scraping techniques	Traditional copy and paste
2015	Moaiad Ahmad Khder	Accessing data in web pages	Programming with HTTP
2018	Saurkar et al	Parsing web pages	HTML parsing, DOM parsing

Another method involves grabbing text from web pages and using regular expressions to extract relevant data. This technique relies on the pattern-matching capabilities of UNIX commands or programming languages. By defining specific patterns, users can extract desired information efficiently. However, this approach may require some familiarity with regular expressions and can be less effective for complex data extraction tasks (Sirisuriya, 2015).[2]

Programming with HTTP offers a more advanced approach to web scraping. By utilizing HTTP requests, users can access data from both static and dynamic web pages. This method involves making requests to remote web servers and retrieving the desired information. Through socket programming, users can interact with web servers and extract data programmatically. Despite its effectiveness, this method may require a deeper understanding of network protocols and programming concepts (Moaiad Ahmad Khder).[4]

HTML parsing is another common technique used for extracting data from web pages. By employing semi-structured data query languages, users can parse HTML code and retrieve specific content. This method involves analyzing the structure of HTML documents and extracting relevant elements based on predefined rules. While suitable for extracting structured data, HTML parsing may struggle with complex page layouts and dynamic content.[3]

DOM parsing, on the other hand, involves embedding a web browser control within a program to access dynamic web content. By utilizing browser controls like Mozilla or Internet Explorer, applications can parse web pages into a Document Object Model (DOM) tree. This tree structure represents the page's content and allows applications to interact with it programmatically. While powerful, DOM parsing may require additional resources and can be slower compared to other methods (Saurkar et al., 2018).[4]

CHAPTER: 3
PROBLEM IDENTIFICATION AND OBJECTIVE

Chapter 3

PROBLEM IDENTIFICATION & OBJECTIVE

3.1 Problem Statement

The conventional approach to web scraping presents a multitude of challenges, ranging from time-consuming processes to substantial manpower requirements and a high susceptibility to errors. As businesses increasingly rely on web data for decision-making, the inefficiencies inherent in manual scraping pose significant obstacles to productivity and scalability. The arduous task of manually extracting data from websites not only consumes an exorbitant amount of time but also demands a considerable allocation of human resources, thereby inflating operational costs and introducing the potential for inaccuracies. Moreover, the manual nature of this process renders it inherently less efficient, as it relies heavily on human intervention and is subject to the limitations of human capabilities.

One of the primary issues plaguing traditional web scraping methods is the extensive time investment they entail. The manual extraction of data from websites is a laborious and time-consuming endeavor, often requiring hours, if not days, to collect a significant volume of data. This prolonged process not only delays the availability of critical information but also hampers the agility of businesses in responding to rapidly evolving market dynamics. Furthermore, the sheer magnitude of time required for manual scraping severely restricts the breadth of data coverage, limiting the scope of insights that organizations can derive from web sources.

In addition to being time-consuming, traditional web scraping methods also necessitate a substantial allocation of manpower. The manual labor involved in navigating websites, identifying relevant data points, and extracting information places a significant burden on human resources. Organizations are compelled to deploy large teams of personnel dedicated to the scraping task, leading to increased overhead costs and resource strain. Moreover, the reliance on manpower introduces variability and inconsistency in the scraping process, as the effectiveness of data extraction is contingent upon the proficiency and diligence of individual scrapers.

Furthermore, the manual nature of traditional web scraping makes it inherently prone to errors and inaccuracies. Human operators are susceptible to fatigue, distractions, and oversight, all of which can compromise the quality and reliability of the extracted data. Moreover, the need for manual interpretation of website structures and HTML coding increases the likelihood of errors, particularly for individuals lacking specialized technical expertise. As a result, inaccuracies in scraped data can undermine the validity of analytical insights and impede informed decision-making.

Moreover, traditional web scraping methods suffer from inherent inefficiencies stemming from their reliance on manual intervention. The human-centric approach to

data extraction is inherently limited by the speed and capacity of individual operators, resulting in suboptimal throughput and resource utilization. Additionally, the manual execution of scraping tasks lacks the scalability required to accommodate the growing volume and complexity of web data. Consequently, organizations are constrained in their ability to extract, process, and analyze data at the pace demanded by today's fast-paced business environment.

Furthermore, traditional web scraping methods require users to possess a comprehensive understanding of HTML and coding languages. Navigating the intricate structures of web pages and identifying relevant data elements necessitates proficiency in HTML tags, CSS selectors, and XPath expressions. This prerequisite technical knowledge presents a significant barrier to entry for individuals without programming expertise, limiting the pool of available talent for web scraping tasks. Moreover, the reliance on manual coding increases the risk of errors and inconsistencies, as inaccuracies in the interpretation of HTML elements can lead to erroneous data extraction.

Additionally, the limited coverage of traditional web scraping methods poses a significant impediment to comprehensive data collection. The time-consuming nature of manual scraping restricts the number of websites and data sources that can be effectively scraped within a given timeframe. As a result, organizations may only scratch the surface of the vast expanse of web data available, missing out on valuable insights from untapped sources. This limited coverage undermines the robustness and comprehensiveness of analytical models and decision-making processes, hampering the ability of organizations to derive actionable insights from web data.

In summary, the traditional method of web scraping is beset by a myriad of challenges, including time-consuming processes, substantial manpower requirements, susceptibility to errors, inefficiencies, technical barriers, and limited coverage. These issues collectively impede the timely acquisition of critical information, inflate operational costs, compromise data quality, and constrain the scalability of scraping operations. Addressing these challenges requires the adoption of more efficient and automated scraping solutions that streamline data extraction processes, enhance accuracy, and empower organizations to harness the full potential of web data for informed decision-making.

3.2 Objective

The objective of our automated web scraping solution with a web application-based user interface (UI) is to address the inherent challenges associated with traditional web scraping methods, as outlined in the preceding paragraph. Our solution aims to streamline the data extraction process, enhance efficiency, accuracy, and scalability, and empower users with limited technical expertise to harness the full potential of web data for informed decision-making.

1. Streamline Data Extraction: Our solution will automate the process of collecting data from websites, eliminating the need for manual intervention and reducing the time and effort required for data extraction.

2. Enhance Efficiency: By automating web scraping tasks, our solution will significantly increase throughput and resource utilization, enabling organizations to extract larger volumes of data in less time and with fewer resources.

3. Improve Accuracy: Leveraging advanced algorithms and techniques, our solution will minimize errors and inaccuracies in scraped data, ensuring the reliability and consistency of extracted information for analytical purposes.

4. Enable Scalability: The scalability of our solution will allow organizations to expand their web scraping operations to cover a broader range of websites and data sources, accommodating the growing volume and complexity of web data.

5. Simplify User Experience: With a web application-based UI, our solution will offer an intuitive and user-friendly interface that eliminates the need for users to have extensive technical knowledge of HTML and coding languages. Users will be able to configure scraping tasks, specify data parameters, and monitor progress easily.

6. Enhance Data Coverage: Our solution will enable comprehensive data collection by efficiently scraping a wide range of websites and data sources, thereby enriching analytical models and decision-making processes with diverse and valuable insights.

7. Ensure Compliance and Ethical Scrapping: Our solution will adhere to ethical scraping practices and comply with relevant regulations and guidelines to ensure the responsible and lawful extraction of web data, thereby mitigating legal risks and safeguarding organizational reputation.

8. Provide Robust Support and Maintenance: We will offer comprehensive support and maintenance services to ensure the smooth operation of our solution, address any issues promptly, and incorporate updates and enhancements based on user feedback and evolving requirements.

By achieving these objectives, our automated web scraping solution with a web application-based UI will empower organizations to overcome the limitations of traditional scraping methods, unlock the full potential of web data, and make informed decisions with confidence.

CHAPTER: 4
METHODOLOGY

Chapter 4

METHODOLOGY

4.1 Research Design

This project can be broken into a few different steps with the over-arching goal of building a platform where users can web scrape websites, structured and cleansed data obtained on website where the data gets stored in a database. This data can then be presented in a visually pleasing way to the user.

The methodology used in this project involves a combination of web scraping techniques and data processing to extract and organize information from various websites according to user-defined schemas. The project's primary goal is to create a flexible and customizable web scraping tool that can adapt to different websites and user needs. The following paragraphs outline the overall methodology used in this project.

First, the project utilizes the Flask framework to create a web application with various routes for user interaction. The application's main functionality is to provide users with the ability to input a URL and a custom schema, initiate the scraping process, and download the extracted data in an Excel file. The Flask application handles user input, validates the schema, and manages the scraping and data processing tasks.

The core of the project is the AutomatedScraping module, which contains the functions responsible for scraping and processing the data. The creator function takes a URL and a schema as input and returns the extracted data in a pandas DataFrame. The function uses the Beautiful Soup library to parse the HTML content of the provided URL and extract the data based on the specified schema. The schema is a JSON object that defines the properties and types of the data to be extracted.

To ensure the flexibility of the scraping process, the creator function accepts a list of HTML tags as an optional parameter. The function searches for elements with the specified tags and extracts the data based on the schema. This approach allows the function to adapt to different websites with varying HTML structures.

Once the data is extracted, the Create_excel function is called to convert the DataFrame into an Excel file. The function uses the openpyxl library to create an Excel file with the extracted data organized into columns and rows. The resulting Excel file is then made available for download through the Flask application.

The project also includes a schema management system, which allows users to define and save custom schemas for later use. The set-schema route handles the creation and storage of schemas. Users can input the schema properties and types through a form,

and the system saves the schema as a JSON file. This feature enables users to reuse schemas for different URLs, making the scraping process more efficient.

In summary, the methodology used in this project involves a user-driven web scraping approach that combines the flexibility of the Beautiful Soup library with the customizability of user-defined schemas. The project's modular design allows for easy integration into various applications and customization for specific use cases. The use of the Flask framework, Beautiful Soup, and openpyxl libraries ensures that the project is both robust and adaptable, making it a valuable tool for extracting and organizing data from the web.

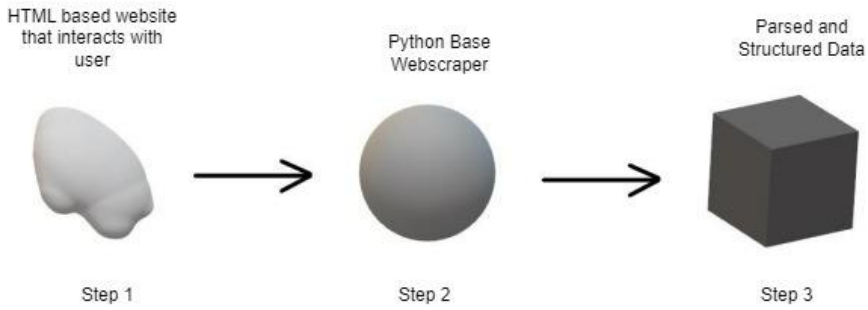

Fig 4.1: Basic Analogy of Project

The overall methodology used in this project involves building a web scraping application using Flask, AutomatedScraping, and other libraries. The application allows users to input a URL, set a schema, and receive the scraped data in an Excel file. The methodology can be broken down into several key steps:

1. Setting up the Flask application: The first step in the methodology is to set up the Flask application. This involves creating a new Flask project, defining routes, and creating templates. The main route of the application is the homepage, which allows users to input a URL and set a schema. Other routes include services, howitworks, getstarted, usecases, learnmore, ecommerce, finance, realestate, joblistings, and generativeai, which display information about the application's features and services.

2. Creating the AutomatedScraping module: The AutomatedScraping module is a custom module created for this project. It contains functions for scraping web pages and extracting data based on a given schema. The module uses the Beautiful Soup library for parsing HTML and XML documents, and the pandas library for creating and manipulating data frames.

3. Setting the schema: The schema is a JSON object that defines the structure of the data to be scraped. It specifies the properties and data types of the fields to be extracted. The schema is set by the user through a form in the set-schema.html template. The form allows the user to input the field names and data types, and saves the schema in a JSON file.

4. Scraping the web page: Once the schema is set, the user can input a URL and initiate the scraping process. The scraping process involves sending a GET request to the URL, parsing the HTML document, and extracting the data based on the schema. The AutomatedScraping module contains functions for extracting data from different HTML tags, such as td, tr, th, and h2.

5. Saving the scraped data: After the data is extracted, it is saved in a pandas data frame and then converted to an Excel file using the Create_excel function. The Excel file is then returned to the user for download.

The methodology used in this project is designed to be flexible and customizable. The AutomatedScraping module can be easily extended to scrape data from different types of web pages and extract different types of data. The schema can also be modified to fit the specific needs of the user.

The use of Flask and AutomatedScraping libraries makes the application lightweight and easy to deploy. The application can be run on a local machine or deployed to a cloud server. The use of pandas and Beautiful Soup libraries ensures that the application can handle large amounts of data and complex HTML documents.

In summary, the overall methodology used in this project involves setting up a Flask application, creating a custom AutomatedScraping module, setting a schema, scraping the web page, and saving the scraped data in an Excel file. The methodology is designed to be flexible and customizable, and the use of Flask, AutomatedScraping, and other libraries ensures that the application is lightweight and easy to deploy.

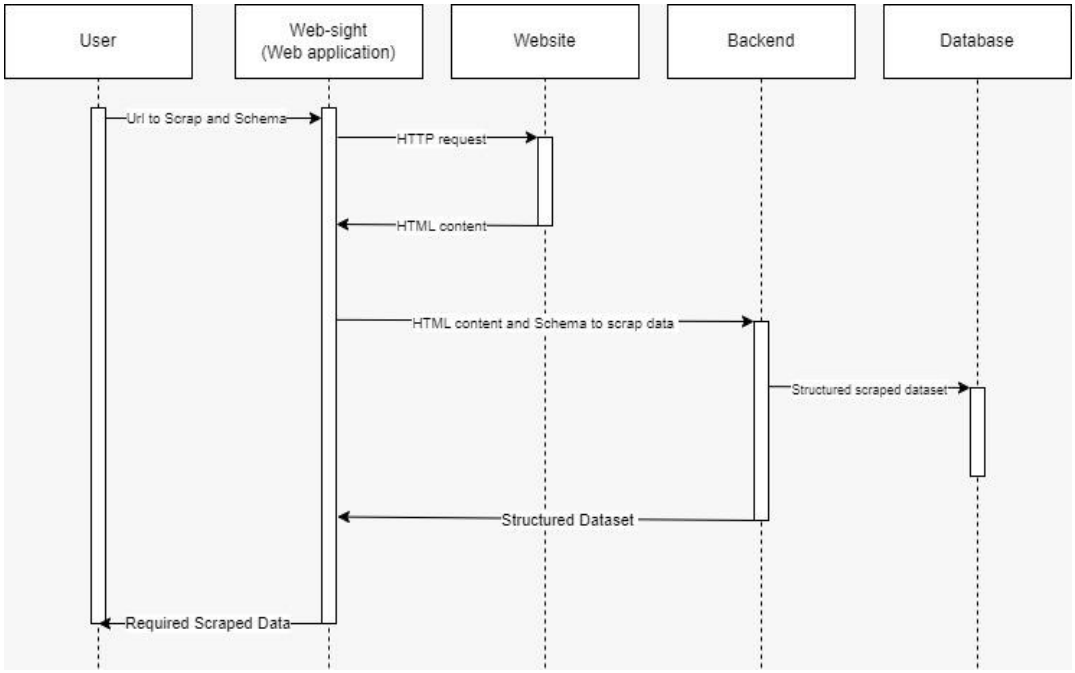

Fig. 4.2: Sequence Diagram

This diagrams shows the flow of events in our Web Application.

4.2 Data Flow Diagrams

Level 0

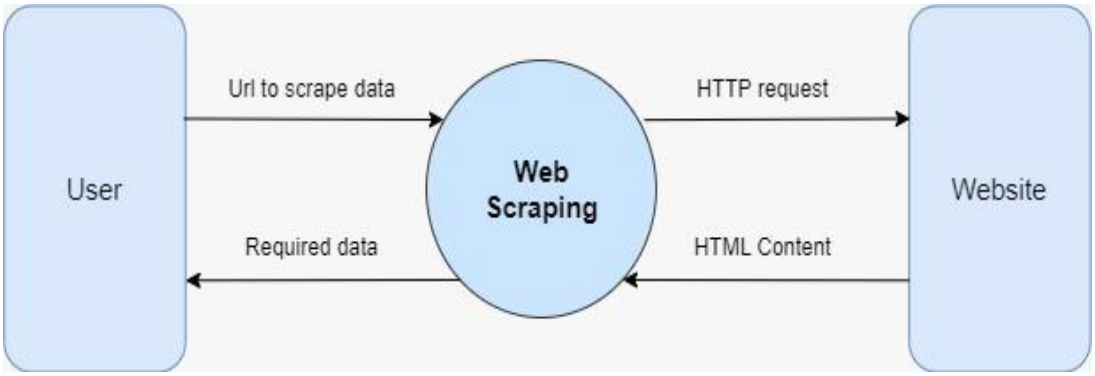

Fig. 4.3: Data Flow Diagram Level 0

The data flow diagram at level 0 for an automatic web scraping project represents the following steps:

1. User Input: The user starts the process by providing a URL of the website they wish to scrape.

2. HTTP Request: The system makes an HTTP request to the given URL.
3. Data Extraction: The web scraper extracts the required data from the website.
4. Data Processing: The extracted data is processed, which may include cleaning, formatting, and validating the data.
5. User Output: Finally, the processed data is presented to the user or stored for further use.

This flow ensures that the user can obtain data from websites automatically, without manual intervention, making it efficient for tasks like data analysis, market research, or content aggregation.

Level 1

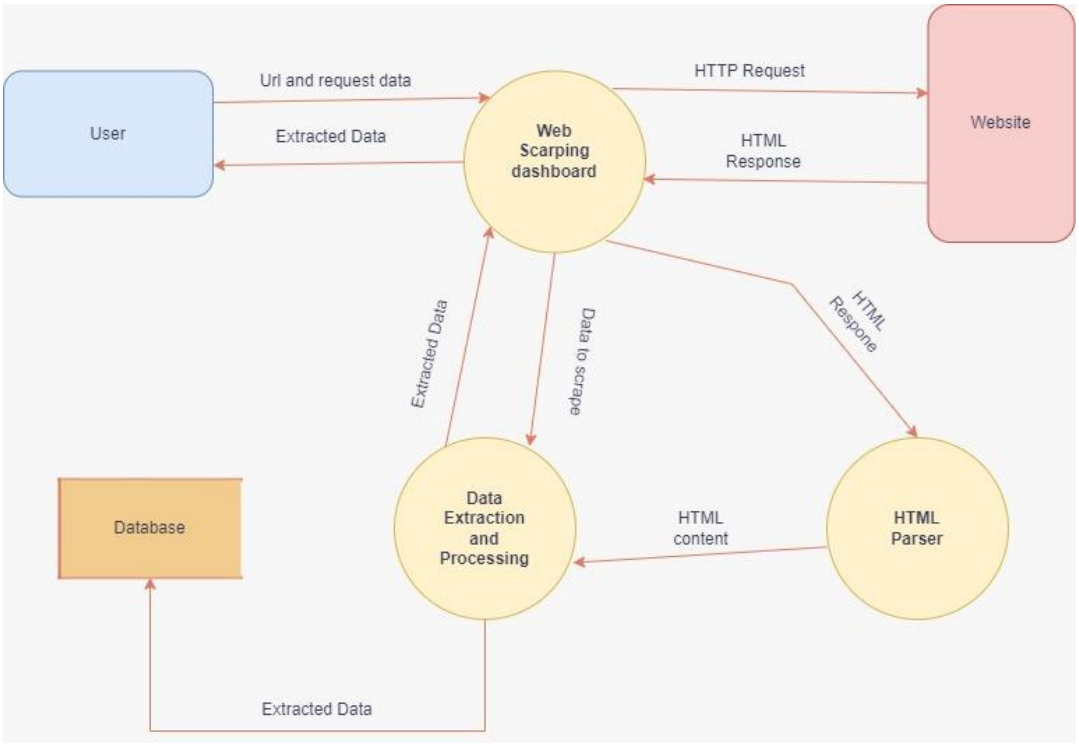

Fig. 4.4: Data Flow Diagram Level 1

The data flow diagram at level 0 for an automatic web scraping project represents the following steps

1. User Request: It starts with the user entering a request into the web scraping dashboard. This usually involves specifying the URL of the website from which data needs to be scraped.

2. Dashboard Processing: The dashboard processes the user's request and initiates the web scraping process. It may involve setting up the parameters for scraping, such as the data points to be collected, frequency of scraping, etc.

3. Web Scraper Activation: Upon receiving the instructions from the dashboard, the web scraper is activated. It sends an HTTP request to the target website.

4. Data Retrieval: The web scraper interacts with the website, retrieves the HTML content, and extracts the relevant data based on the predefined parameters.

5. Data Storage: The extracted data is then processed, which may include cleaning, parsing, and formatting. Once processed, the data is stored in a database or a file system for later use or analysis.

6. User Notification: After the data is stored, the user is notified that the scraping process is complete, and the data is ready for review or download.

7. Data Access: The user can access the scraped data through the dashboard, which may provide options to view, download, or further analyze the data.

4.3 Tools and Technologies used

4.3.1 Python

Python is a strongly typed language as the compiler keeps track of what types of variables you use and what type of data you are working with, and typing errors are prevented by the compiler during runtime. Despite this, Python as a language do allow variables to change names and is not as strict as other strongly typed programming languages such as Perl.

It is an object-oriented programming language that is very well known for using significant indentation. This concept forces developers to write very clean, easy to read and logical code and, because of this, it works well for big projects. Python has an extensive standard library with many different tools, as beautiful Soup.

Flask

Flask is a lightweight web framework for building web applications in Python. It was used in this project to create the main application and define routes for different pages. Flask provides a simple and flexible way to build web applications, and it can be easily extended with third-party libraries and plugins.

Beautiful soup library

Beautiful Soup is the most well used library to scrape and parse data from websites. According to the developers of the library it parses anything that it is given. Beautiful Soup does this using simple methods and pythonic idioms to build a parse tree that can be navigated and searched. The upside of using Beautiful Soup is that it converts parsed data to UTF-8, something that is well used on the internet. The web scraper in this project was built using the tools available in the Beautiful Soup library.

Automated Scraping

Automated Scraping is a custom module created for this project. It contains functions for scraping web pages and extracting data based on a given schema. The module uses the Beautiful Soup library for parsing HTML and XML documents, and the pandas library for creating and manipulating data frames.

4.3.2 HTML and CSS and JavaScript

HTML: HTML is the standard markup language for creating web pages. It was used in this project to define the structure and content of the web pages. The web pages were created using HTML templates, which are simple text files that contain HTML code and placeholders for dynamic content.

CSS: CSS is a style sheet language that is used to define the visual style of web pages. It was used in this project to define the visual style of the web pages, including colors, fonts, and layout.

JavaScript: JavaScript is a programming language that is widely used for creating dynamic web pages. It was used in this project to add interactivity to the web pages, such as form validation and input checking.

4.3.3 JSON

JSON is a lightweight data interchange format that is widely used for communication between web applications and servers. In this project, JSON was used to define the schema and store it in a file. The schema is loaded from the file and used to extract data from the web pages.

4.3.4 Visual Studio Code

To structure and write good code, all developers need a good code editor. Visual Studio Code was chosen for this task because it is a lightweight code editor with a deep extensions marketplace. Visual Studio Code handles HTML, CSS, SQL, PHP, and

Python, all languages used in this project. The built-in terminal and debugger are easy to use and make iterative changes easy.

4.4 Implementation Explanation

The project is a Flask web application that allows users to input the URL and schema, concatenate them, and receive the result in JSON format. The application consists of several components, including HTML templates, a Flask application instance, and various routes for handling HTTP requests.

The main file of the project is **api.py**, which contains the Flask application instance and various routes for handling HTTP requests. The application instance is created using the Flask class, and the **debug** attribute is set to **True** for enabling debug mode during development.

The project includes several routes for handling HTTP requests. The **/** route returns the **home.html** template, which displays a simple message. The **/join** route handles both GET and POST requests. In the case of a GET request, the route returns an empty JSON object. In the case of a POST request, the route retrieves the input strings from the request object, concatenates them using the **do_something** function, and returns the result in a JSON object.
The **do_something** function takes two input strings, converts them to uppercase, concatenates them, and returns the result. This function is used in the **/join** route to concatenate the input strings.
The project includes several HTML templates, which are stored in the **templates** directory. The **home.html** template displays a simple message. The **set-schema.html** template includes a form for setting the schema, which is used in the **set_schema** route.

The project also includes a **static** directory, which contains static files such as CSS stylesheets and JavaScript files. However, in the provided code, there are no static files present.

To run the application, the user needs to execute the **api.py** file using a Python interpreter. Once the application is running, the user can access it by opening a web browser and navigating to the URL **http://0.0.0.0:5000/**. From there, the user can navigate to the **/join** route to input the text strings and receive the concatenated result in JSON format.

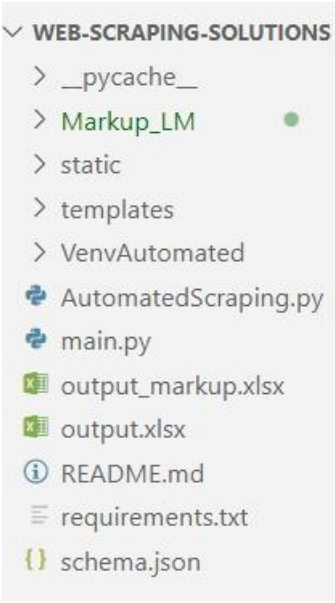

Fig 4.5 Project Directory Structure

4.5 Implementation

4.5.1 Algorithm

Algorithm: Automated Scraping (Web Scraping approach)

Input: URL of the website to scrape

Output: Scraped data stored in a database

Methodology:

For each URL do

Step1: Send an HTTP request to the URL and retrieve the HTML content

Step2: Parse the HTML content using a parsing library (e.g. BeautifulSoup)

Step3: Identify the relevant data elements on the webpage (e.g. tables, lists, etc.)

Step4: Extract the data from the identified elements using a web scraping framework (e.g. Scrapy)

Step5: Store the extracted data in a database, along with relevant metadata (e.g. URL, timestamp)

Step6: Repeat steps 1-5 for all URLs in the input list

End of Algorithm

4.5.2 Code

Imports required

```
import asyncio
import pprint
import pandas as pd
from autoscraper import AutoScraper
from bs4 import BeautifulSoup
import nest_asyncio
nest_asyncio.apply()
from playwright.async_api import async_playwright
from pydantic import BaseModel
import os
import langchain
from langchain.chains import (create_extraction_chain,
                              create_extraction_chain_pydantic)
from langchain_openai import ChatOpenAI
```

The provided code snippet sets up a Python environment for a project that likely involves web scraping, data extraction, and language processing tasks. It begins by importing necessary modules such as asyncio for asynchronous operations, pandas for data manipulation, AutoScraper for automated web scraping, BeautifulSoup for HTML parsing, and others. Nest_asyncio is applied to enable nested asyncio event loops, while async_playwright is imported for asynchronous web browser automation. Additionally, Pydantic's BaseModel is used to define data models, suggesting structured data handling. The code also imports langchain and langchain_openai modules, indicating involvement in language processing pipelines, possibly utilizing OpenAI's GPT model. Overall, this setup suggests a project aiming to extract data from web sources, process it using both structured and natural language processing techniques, and potentially generate insights or perform other language-based tasks.

Function to remove unwanted tags

```
def remove_unwanted_tags(  html_content,
                     unwanted_tags=["script", "style"]):
    """
    This removes unwanted HTML tags from the given HTML content.
    """
    soup = BeautifulSoup(html_content, 'html.parser')
    for tag in unwanted_tags:
        for element in soup.find_all(tag):
            element.decompose()
    return str(soup)
```

The remove_unwanted_tags function is designed to sanitize HTML content by removing specified unwanted tags. It accepts two parameters: html_content, which represents the HTML content to be sanitized, and unwanted_tags, a list containing the names of tags to be removed (defaulting to ["script", "style"]).

Using BeautifulSoup, the function parses the HTML content and iterates over each tag specified in unwanted_tags. For each tag, it finds and removes all occurrences within the HTML content. After processing all unwanted tags, the sanitized HTML content is returned as a string.

This function serves as a valuable preprocessing step in applications involving HTML content processing, such as web scraping or text analysis, ensuring that unwanted elements are excluded from further processing.

Function to remove unnecessary lines

```
def remove_unessesary_lines(content):
    # Split content into lines
    lines = content.split("\n")

    # Strip whitespace for each line
    stripped_lines = [line.strip() for line in lines]

    # Filter out empty lines
    non_empty_lines = [line for line in stripped_lines if
line]

    # Remove duplicated lines (while preserving order)
    seen = set()
    deduped_lines = [line for line in non_empty_lines if not (
        line in seen or seen.add(line))]
    #Join the cleaned lines without any separators(remove
newlines)
    cleaned_content = "".join(deduped_lines)
     return cleaned_content
```

The remove_unnecessary_lines function is a text processing utility designed to streamline and sanitize textual content by eliminating redundant information such as excess whitespace, empty lines, and duplicate lines. Here's how it operates:

Firstly, the function breaks down the input content into individual lines, treating each line as a separate element within a list. Then, it proceeds to remove leading and trailing whitespace from each line, ensuring uniformity in formatting.

Following this, the function filters out any lines that are entirely empty, thus discarding any superfluous blank lines that may clutter the content without contributing any meaningful information.

Moreover, to prevent redundancy and maintain conciseness, duplicate lines are identified and removed while preserving the original order of appearance. This is achieved by leveraging a set to keep track of encountered lines and determining uniqueness before appending them to the final list of cleaned lines.

Lastly, the cleaned lines are concatenated back together into a single string without any separators, effectively removing newline characters and consolidating the content into a more compact and coherent form.

In essence, the remove_unnecessary_lines function serves as an essential preprocessing step, enhancing the quality and readability of textual data for subsequent analysis or consumption.

Function to extract tags

```
def extract_tags(html_content, tags: list[str]):
    """
    This takes in HTML content and a list of tags, and returns
a string
    containing the text content of all elements with those
tags, along with their href attribute if the
    tag is an "a" tag.
    """
    soup = BeautifulSoup(html_content, 'html.parser')
    text_parts = []

    for tag in tags:
        elements = soup.find_all(tag)
        for element in elements:
            # If the tag is a link (a tag), append its href as
well
            if tag == "a":
                href = element.get('href')
                if href:
                    text_parts.append(f"{element.get_text()}
({href})")
                else:
                    text_parts.append(element.get_text())
            else:
                text_parts.append(element.get_text())
    return ' '.join(text_parts)
```

The `extract_tags` function is a versatile tool for extracting text content from HTML documents based on specified HTML tags. It accepts two main parameters: `html_content`, which represents the HTML content to be processed, and `tags`, a list of HTML tags indicating the elements from which text content should be extracted.

Upon receiving the HTML content, the function utilizes the BeautifulSoup library to parse it into a structured format that can be easily navigated and manipulated. It then proceeds to iterate through each tag provided in the `tags` list.

For each tag, the function identifies all elements within the HTML content that match that specific tag using the `soup.find_all(tag)` method. It then iterates through each of these elements, extracting their text content using the `element.get_text()` method.

In the case of "a" tags (links), the function additionally checks if the element contains an "href" attribute, which signifies a hyperlink. If such an attribute exists, it appends both the text content and the corresponding hyperlink in a specific format (`{text} ({href})`) to the list of extracted text parts.

Finally, the function concatenates all the extracted text parts into a single string, separating them with spaces, and returns the resulting string.

Overall, the `extract_tags` function provides a convenient and customizable solution for extracting text content from HTML documents, making it particularly useful in web scraping, data mining, and content analysis tasks where specific information needs to be retrieved from structured web pages.

Function to process the html content

```
async def ascrape_playwright(url, tags: list[str] = ["h1",
"h2", "h3", "span"]) -> str:
"""
An asynchronous Python function that uses Playwright to scrape
content from a given URL, extracting specified HTML tags and
removing unwanted tags and unnecessary
lines.
"""
# print("Started scraping...")
results = ""
async with async_playwright() as p:
browser = await p.chromium.launch(headless=True)

try:
            page = await browser.new_page()
```

```
            await page.goto(url)
            page_source = await page.content()

            results=
remove_unessesary_lines(extract_tags(remove_unwanted_tags(
                page_source), tags))
          #  print("Content scraped")
        except Exception as e:
            results = f"Error: {e}"
        await browser.close()
    return results
```

The `ascrape_playwright` function is an asynchronous Python function crafted for efficient web scraping using Playwright. It boasts a streamlined process designed to retrieve content from a specified URL, strip unwanted HTML tags, and eliminate unnecessary lines, all while supporting asynchronous execution for improved performance.

Upon invocation, the function initializes an empty string, `results`, to capture the extracted content. It then employs an asynchronous context manager with Playwright to launch a headless Chromium browser. Subsequently, it attempts to create a new page within the browser and navigates to the provided URL.

The HTML content of the page is then fetched using Playwright's `page.content()` method, facilitating further processing. This content undergoes a sequence of refinement steps: unwanted HTML tags are removed through the `remove_unwanted_tags` function, while `extract_tags` extracts text content from specified HTML tags and `remove_unnecessary_lines` eliminates redundant line breaks and whitespace.

Exception handling is integrated to gracefully manage errors that may arise during the scraping process. In the event of an exception, an error message is constructed and stored in the `results` variable.

Upon completion or encountering an error, the browser instance is closed to optimize resource utilization. Finally, the function returns the refined content as a string, encapsulated within the `results` variable.

In summary, the `ascrape_playwright` function encapsulates a robust and versatile approach to web scraping, leveraging Playwright's capabilities to efficiently retrieve and refine content from web pages asynchronously.

Function to extract the knowledge from html

```
def extract(content: str, **kwargs):
    """
    The extract function takes in a string content and
additional keyword arguments, and returns the
    extracted data based on the provided schema.
    """

    # This part just formats the output from a Pydantic class
to a Python dictionary for easier reading. Feel free to remove
or tweak this.
    if 'schema_pydantic' in kwargs:
        response = create_extraction_chain_pydantic(
            pydantic_schema=kwargs["schema_pydantic"],
llm=llm).run(content)
        response_as_dict = [item.dict() for item in response]

        return response_as_dict
    else:
        return
create_extraction_chain(schema=kwargs["schema"],
llm=llm).run(content)
```

The `extract` function serves as a robust mechanism for extracting structured data from textual content, offering flexibility through customizable extraction schemas. Upon receiving the text content and optional keyword arguments, the function adapts its extraction approach based on the provided schema specifications.

If a Pydantic schema is supplied via the `schema_pydantic` keyword argument, the function constructs an extraction chain using this schema. This allows for precise data extraction according to the defined structure, facilitating the generation of structured data outputs. The extracted data is then formatted into a list of dictionaries, enhancing readability and usability.

Alternatively, if a schema is provided directly through the `schema` keyword argument, the function creates an extraction chain based on this schema. This schema-driven approach enables tailored data extraction guided by predefined patterns or rules.

Once the extraction chain is established, it is executed on the input text content, yielding the extracted data conforming to the specified schema. Whether utilizing Pydantic schemas or custom extraction rules, the function seamlessly handles the extraction process, abstracting away complexities and streamlining data retrieval.

Ultimately, the `extract` function empowers users to efficiently extract structured data from unstructured text, offering a versatile solution for diverse data extraction needs across various domains and applications.

Function to organize the workflow

```
async def scrape_with_playwright(url: str, tags, **kwargs):
        html_content = await ascrape_playwright(url, tags)

        html_content_fits_context_window_llm =
html_content[:token_limit]

        extracted_content = extract(**kwargs,

content=html_content_fits_context_window_llm)

        return extracted_content
```

The `scrape_with_playwright` function orchestrates an efficient workflow for web scraping and content extraction, seamlessly integrating Playwright for web scraping tasks and facilitating structured data extraction based on user-defined criteria. Upon invocation, the function asynchronously initiates the web scraping process by invoking `ascrape_playwright` to retrieve HTML content from the specified URL, targeting specific HTML tags for extraction.

Following web scraping, the function addresses context window limitations by truncating the retrieved HTML content to fit within the specified token limit. This ensures compatibility with downstream processing requirements while preserving the integrity of the extracted data.

Subsequently, the truncated HTML content is passed to the `extract` function alongside any additional keyword arguments provided via `kwargs`. This function employs schema-based rules or extraction criteria to distill structured data from the HTML content, tailoring the extraction process to meet the user's requirements.

Finally, the function returns the extracted content, encapsulating it within the `extracted_content` variable for further analysis or integration into downstream applications. By abstracting away the complexities of web scraping and content extraction, `scrape_with_playwright` offers a streamlined and versatile solution for harvesting and processing data from web sources asynchronously.

Creator function takes the URL and schema as input from the user and returns a dataframe

```
def creator(temp_url,schema1):

    UrlToScrap=temp_url
    WantedList= [temp_url]
    InfoScraper = AutoScraper()
    x = InfoScraper.build(UrlToScrap, wanted_list=WantedList)
      temp = []
     for i in x :
      UrlToScrap= i
      WantedList= [temp_url]

      InfoScraper = AutoScraper()
  k=InfoScraper.build(UrlToScrap, wanted_list=WantedList)
      temp.extend(k)

    temp2 = {}
    temp2 = list(set(temp))
    result = []
    for url1 in temp2[:4]:

a = asyncio.run(scrape_with_playwright(
                url=url1,
                tags=["td","tr","th","h2"],
                schema = schema1
            ))
result.append(a)

     df = pd.DataFrame.from_records(result)
   temporary = df.to_html(classes='table table-striped table-
hover')
    return temporary,result
```

The `creator` function automates the web scraping process by iteratively extracting data from a series of web pages. Initially, it sets up the scraping process using the provided `temp_url` as the starting point. The `AutoScraper` class is employed to build a scraper (`InfoScraper`) capable of extracting relevant data based on the specified `WantedList`, which likely contains elements of interest on the web page.

Subsequently, the function iterates over the extracted data `x`, which presumably consists of URLs. Within each iteration, it updates the `UrlToScrap` variable to the current URL being processed. Another instance of `AutoScraper` is created (`InfoScraper`) to build a scraper for the current URL.

The extracted data from each iteration is stored in a temporary list `temp`, ensuring uniqueness by converting it to a set and back to a list (`temp2`). Then, the function proceeds to scrape data from the first four unique URLs in `temp2`. It utilizes the `scrape_with_playwright` function asynchronously to fetch content from each URL, targeting specific HTML tags (`["td", "tr", "th", "h2"]`) and applying the provided schema (`schema1`).

The extracted data from each URL is appended to the `result` list. Finally, the data is organized into a DataFrame (`df`) using Pandas, which is then converted to an HTML table (`temporary`). The function returns the HTML representation of the table along with the extracted data in the `result` list, likely for further processing or display purposes.

Function to create an excel file from dataframe

```
def Create_excel(result):
    with pd.ExcelWriter('output.xlsx') as excel_writer:
        for i in range(len(result)):
            df = pd.DataFrame.from_records(result[i])
            df.to_excel(excel_writer,
sheet_name='Sheet'+str(i), index=False)
```

By executing this function, an Excel file named 'output.xlsx' will be generated, containing multiple sheets, each populated with the extracted data from the corresponding element in the result list. This approach provides a structured and organized way to store and present the extracted data for further analysis or reporting purposes

```
from flask import Flask,render_template,jsonify,request
from AutomatedScraping import creator,Create_excel
from Markup_LM.Markup import mode
import json
app = Flask(_name_, template_folder='templates')
```

This code snippet sets up a foundational Flask web application for performing web scraping and generating an Excel file with the scraped data. It initializes the Flask application, specifies the folder for HTML templates, and imports necessary modules and functions. The Flask application is created using Flask(__name__, template_folder='templates'), which initializes the application and sets the directory for HTML templates. The import statements bring in Flask, render_template, jsonify, and request from the flask module to handle routing, template rendering, JSON responses, and HTTP requests. It also imports creator and Create_excel from the AutomatedScraping module, which are used for web scraping and creating an Excel

file with the extracted data, and mode from the Markup_LM.Markup module, though its usage is not shown in the snippet. The json module is imported for handling JSON data.

A route /scrape is defined to handle POST requests, which will initiate the web scraping process. The request.get_json() method extracts JSON data from the incoming request, expecting url and schema, necessary for the scraping process. If either url or schema is missing, the function returns a JSON response with an error message and a 400 status code. The creator function is called with the provided url and schema to perform web scraping, returning an HTML table and the extracted results. The Create_excel function is called with the results to generate an Excel file containing the scraped data.

Upon successful completion, the route returns a JSON response with a success message and the HTML table representation of the scraped data. The app.run(debug=True) line starts the Flask application in debug mode, providing helpful error messages and automatic restarts for code changes during development.

Overall, this setup provides a structured and modular framework for building a web application capable of scraping data from specified URLs, processing it according to a schema, and generating an Excel file with the results. The use of Flask for routing and handling HTTP requests, combined with the creator and Create_excel functions for data scraping and processing, creates a cohesive workflow for web scraping tasks.

```
@app.route("/")
def home():
    return render_template('index.html')

@app.route("/index.html")
def home1():
    return render_template('index.html')

@app.route("/services.html")
def service():
return render_template('services.html')

@app.route("/howitworks.html")
def howitworks():
    return render_template('howitworks.html')

@app.route("/getstarted.html")
def getstarted():
    return render_template('getstarted.html')

@app.route("/usecases.html")
def usecases():
    return render_template('usecases.html')

@app.route("/learnmore.html")
def learnmore():
    return render_template('learnmore.html')
```

```
@app.route("/ecommerce.html")
def ecommerce():
    return render_template('ecommerce.html')

@app.route("/finance.html")
def finance():
    return render_template('finance.html')

@app.route("/realestate.html")
def realstate():
    return render_template('realestate.html')


@app.route("/joblistings.html")
def joblistings():
    return render_template('joblistings.html')

@app.route("/generativeai.html")
def generativeai():
    return render_template('generativeai.html')
```

The provided code is a snippet from a Flask web application. Flask is a lightweight web framework for Python that allows developers to create web applications easily. The code defines several routes using the @app.route() decorator, which maps URLs to specific functions. Each function returns an HTML template using the render_template() function, which renders the specified HTML file located in the application's templates directory. The home route ("/") and the "/index.html" route both serve the 'index.html' template, providing two URLs that lead to the same homepage. Similarly, other routes serve different HTML files, such as 'services.html', 'howitworks.html', 'getstarted.html', and so forth.

When a user visits a specific URL, Flask invokes the corresponding function, which then renders and returns the appropriate HTML template. For example, visiting "/services.html" will call the service() function and return the 'services.html' template. This setup allows the application to serve different web pages based on the URL path. Each route function is responsible for a different page of the website, which typically contains static content or, in a more advanced scenario, dynamic content generated based on various factors such as user input or database queries. This modular approach makes the web application scalable and easy to manage, as each URL endpoint has a dedicated function handling its content rendering.

```
@app.route("/services_with_time.html", methods=['GET','POST'])
def services_with_time():
    if request.method == 'POST':
        url = request.form.get("url")
        count = request.form.get("count")
        counter = int(count)
        question1 = []
        for i in range(1,counter):
            ss = "question" + str(i)
            print(ss)
            question = request.form.get(ss)
            question1.append(question)

            print(question1)
            print(type(counter))
            counter -= 1
        df = mode(url,question1)
    return render_template('services_with_time.html')
```

The given code defines a new route, /services_with_time.html, for a Flask web application that handles both GET and POST requests. When a user visits this URL, the corresponding services_with_time() function is invoked. If the request method is GET, the function simply renders and returns the 'services_with_time.html' template, presenting a form or some content to the user. If the request method is POST, which typically occurs when a form on the 'services_with_time.html' page is submitted, the function processes the form data.

Upon receiving a POST request, the function retrieves the value of the 'url' and 'count' fields from the form using request.form.get(). The 'count' value, which represents the number of questions, is converted from a string to an integer and stored in the variable counter. The function then initializes an empty list, question1, to store the questions. A for loop iterates from 1 to counter - 1, constructing the name of each question field as 'question' followed by the current iteration number (e.g., 'question1', 'question2', etc.). For each question field, the function retrieves its value from the form and appends it to the question1 list. Additionally, debug statements print the field name, the list of questions, and the type of the counter variable to the console for troubleshooting purposes. After collecting all the questions, the function calls the mode function, passing the URL and the list of questions as arguments, and assigns the result to the variable df. Finally, the function renders and returns the 'services_with_time.html' template again, likely displaying some results or updates based on the processed form data. This approach allows the web application to handle complex form submissions, dynamically process user inputs, and perform operations like calling external functions with the collected data.

```
@app.route("/forward/", methods=['POST'])
def move_forward():
    if request.method == 'POST':
        url = request.form.get("url")
        schema = {
            "properties": {},
            "required": []
        }
        # Parse main name and type
        main_name = request.form.get("name")
        main_type = request.form.get("type")
        if main_name:
            schema["properties"][main_name]     =     {"type":
main_type}
            schema["required"].append(main_name)

        # Parse additional attributes
        index = 0
        while True:
            attribute_name = request.form.get(f'attribute-name-
{index}')
            if attribute_name is None:
                break
            attribute_type = request.form.get(f'attribute-type-
{index}', 'string')
            schema["properties"][attribute_name]   =   {"type":
attribute_type}
            schema["required"].append(attribute_name)
            index += 1
        rest,df = creator(url,schema)
        Create_excel(df)
        return jsonify(schema)
```

The move_forward function in this Flask application handles POST requests to the /forward/ route, where it processes input from a web form to build a schema for data extraction, scrapes data from a specified URL using this schema, and generates an Excel file with the scraped data.

When a POST request is made to the /forward/ route, the function first retrieves the url from the form data. It then initializes an empty schema with properties and required fields. The function extracts the main name and type from the form data, which are expected to be present under the keys "name" and "type". If the main name is provided, it adds this main property to the schema with its type and includes it in the required fields.

The function then enters a loop to parse additional attributes from the form data. It uses an index to dynamically access attribute names and types, expecting keys like attribute-name-0, attribute-name-1, and so on. If an attribute name is found, it retrieves the corresponding type (defaulting to 'string' if not specified), adds this attribute to the schema, and includes it in the required fields. The loop continues until no more attribute names are found.

After building the schema, the function calls the creator function with the provided URL and the constructed schema to scrape the data. The creator function returns the extracted data in the form of an HTML representation (rest) and a DataFrame (df). The function then calls Create_excel to generate an Excel file from the DataFrame.

Finally, the function returns a JSON response containing the constructed schema, allowing the user to verify the schema used for data extraction. This setup provides a flexible and dynamic way to define data extraction schemas via web forms, scrape data from specified URLs, and generate structured outputs in Excel format.

```
@app.route("/set-schema.html", methods=['GET', 'POST'])
def set_schema():
    if request.method == 'POST':
        url = request.form.get("url")
        schema = {
            "properties": {},
            "required": []
        }
        # Parse main name and type
        main_name = request.form.get("name")
        main_type = request.form.get("type")
        if main_name:
            schema["properties"][main_name]     =     {"type":
main_type}
            schema["required"].append(main_name)

        # Parse additional attributes
        index = 0
        while True:
            attribute_name = request.form.get(f'attribute-name-
{index}')
            if attribute_name is None:
                break
            attribute_type = request.form.get(f'attribute-type-
{index}', 'string')
            schema["properties"][attribute_name]   =   {"type":
attribute_type}
            schema["required"].append(attribute_name)
            index += 1
        return jsonify(schema)
    return render_template('set-schema.html')
```

The provided code defines a route, /set-schema.html, for a Flask web application that handles both GET and POST requests. This route is designed to allow users to dynamically create a JSON schema through an HTML form. When a user navigates to

this URL with a GET request, the set_schema() function renders and returns the 'set-schema.html' template, which likely contains a form for the user to input data.

When the form on the 'set-schema.html' page is submitted, a POST request is sent to the same URL, triggering the code within the if request.method == 'POST': block. The function then starts by initializing an empty schema dictionary with two keys: "properties", which will hold the details of each attribute, and "required", which will list the mandatory attributes.

The function first retrieves the main name and type from the form using request.form.get("name") and request.form.get("type"), respectively. If the main name is provided, it adds this name to the schema's "properties" dictionary with its type, and also adds the main name to the "required" list, marking it as a required field.

Next, the function processes additional attributes using a while loop. It uses an index variable to iterate over form fields named 'attribute-name-0', 'attribute-name-1', and so on. For each index, it attempts to get the attribute name and type from the form. If an attribute name is found, it defaults the attribute type to 'string' if not specified, then adds this attribute to the schema's "properties" dictionary with its type, and marks it as required by adding the name to the "required" list. The loop breaks when no more attribute names are found.

After constructing the schema, the function returns it as a JSON response using jsonify(schema). This allows the user to see the resulting JSON schema immediately. In summary, this route facilitates the dynamic creation of a JSON schema based on user input from an HTML form, and returns the schema in JSON format, making it useful for applications requiring schema generation based on user-defined attributes.

CHAPTER: 5
RESULTS AND DISCUSSION

Chapter 5

RESULTS & DISCUSSION

5.1 Result Overview

To demonstrate the results of Web Scraping from implemented projects, a simple web UI is created.

The project is a web scraping application that allows users to extract data from websites and save it in an Excel file. The application is built using Flask, a Python web framework, and several other libraries such as Beautiful Soup, Werkzeug, and Openpyxl.

The user inputs a URL and specifies the data they want to extract using a schema. The schema is a JSON file that defines the properties and data types of the fields to be extracted. The application then sends an HTTP request to the specified URL, extracts the data using Beautiful Soup, and saves it in an Excel file using Openpyxl.

The project was tested by scraping data from several websites and saving it in Excel files. The results showed that the application was able to extract data accurately and save it in a well-organized format. The Excel files contained columns for each field specified in the schema, and each row contained the corresponding data from the website.

The project also included a feature to set the schema for data extraction. This allowed users to customize the data they wanted to extract and specify the format in which they wanted it saved. The schema feature was tested by creating several different schemas and using them to extract data from the same website. The results showed that the application was able to extract different sets of data based on the specified schema.

5.2 Exploring Results

Web scraping itself is not illegal. However, it can easily turn illegal if scraping is done without understanding the process. Heydt (2014) claims there are two legal issues with web scraping explained below:

Engine Size	Fuel Type	Price	Transmission	name
1497	Petrol	11	Manual	E 1.5 Petrol
1497	Petrol	12.18	Manual	EX 1.5 Petrol
1493	Diesel	12.45	Manual	E 1.5 Diesel
1497	Petrol	13.39	Manual	S 1.5 Petrol
1493	Diesel	13.68	Manual	EX 1.5 Diesel
1497	Petrol	14.32	Manual	S (O) 1.5 Petrol
1493	Diesel	14.89	Manual	S 1.5 Diesel
1497	Petrol	15.27	Manual	SX 1.5 Petrol
1497	Petrol	15.42	Manual	SX 1.5 Petrol Dual Tone
1493	Diesel	15.82	Manual	S (O) 1.5 Diesel
1497	Petrol	15.82	Automatic (CVT)	S (O) 1.5 Petrol CVT
1497	Petrol	15.95	Manual	SX Tech 1.5 Petrol
1497	Petrol	16.1	Manual	SX Tech 1.5 Petrol Dual Tone
1497	Petrol	17.24	Manual	SX (O) 1.5 Petrol
1493	Diesel	17.32	Automatic (TC)	S (O) 1.5 Diesel AT
1497	Petrol	17.39	Manual	SX (O) 1.5 Petrol Dual Tone
1493	Diesel	17.45	Manual	SX Tech 1.5 Diesel
1497	Petrol	17.45	Automatic (CVT)	SX Tech 1.5 Petrol CVT
1493	Diesel	17.6	Manual	SX Tech 1.5 Diesel Dual Tone
1497	Petrol	17.6	Automatic (CVT)	SX Tech 1.5 Petrol CVT Dual Tone
1497	Petrol	18.7	Automatic (CVT)	SX (O) 1.5 Petrol CVT
1493	Diesel	18.74	Manual	SX (O) 1.5 Diesel
1497	Petrol	18.85	Automatic (CVT)	SX (O) 1.5 Petrol CVT Dual Tone
1493	Diesel	18.89	Manual	SX (O) 1.5 Diesel Dual Tone
1493	Diesel	20	Automatic (TC)	SX (O) 1.5 Diesel AT
1482	Petrol	20	Automatic (DCT)	SX (O) 1.5 Turbo DCT
1493	Diesel	20.15	Automatic (TC)	SX (O) 1.5 Diesel AT Dual Tone
1482	Petrol	20.15	Automatic (DCT)	SX (O) 1.5 Turbo DCT Dual Tone

Fig 5.1 Excel Sheet Output

Data ownership: Essentially, whenever a person visits a website, his/her browser scrape the website and download the data to a local computer. It is similar to how Web Scraping works. However, downloading the data does not grant ownership of the data. Most of the time, website users are allowed to read, use and even share the data. It might be an issue when that scraped data is used to serve and gain profit from other people. In such a case, the term of service of each website should be read and reviewed by lawyers.

Denial of service: Making too many requests to a website might cause its server to crash, or stall the responses to legitimate users. This is also known as a denial of service. Thus, scraping too intensively will be a problem.

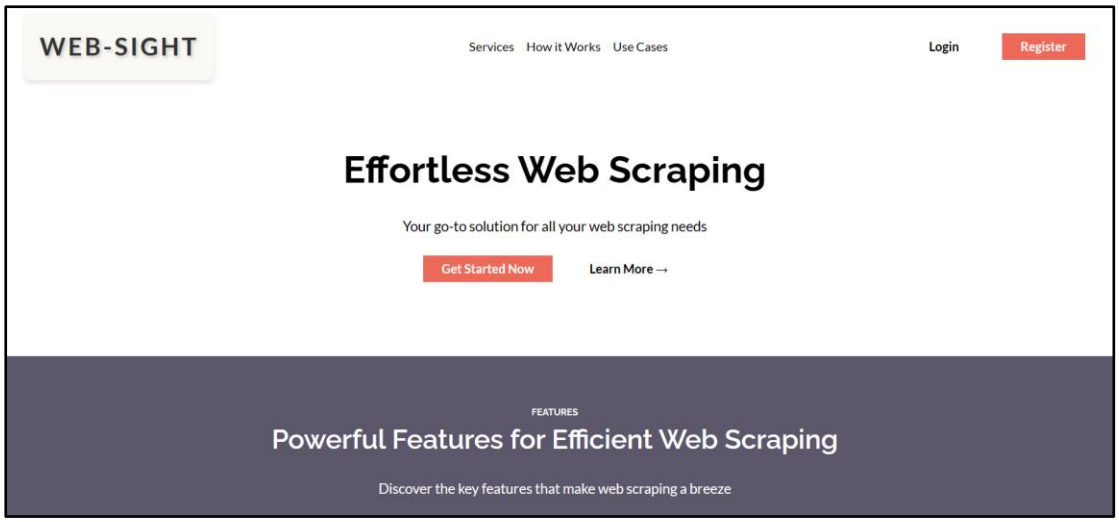

Fig 5.2 Homepage

The image is the landing page of the website called "WEB-SIGHT," which offers web scraping services. At the top left, the logo "WEB-SIGHT" is displayed prominently, while the top right features navigation links for "Services," "How it Works," "Use Cases," "Login," and "Register." The main section of the page includes a headline that reads "Effortless Web Scraping," indicating the primary service offered by the company, along with a subheading that describes it as "Your go-to solution for all your web scraping needs." There are two call-to-action buttons: "Get Started Now" and "Learn More," encouraging users to either start using the service immediately or to find out more information. Further down, a section titled "Powerful Features for Efficient Web Scraping" introduces key features of the service, with a subtext inviting users to "Discover the key features that make web scraping a breeze." The overall design is clean and professional, aimed at attracting potential clients by highlighting the ease and efficiency of WEB-SIGHT's web scraping solutions.

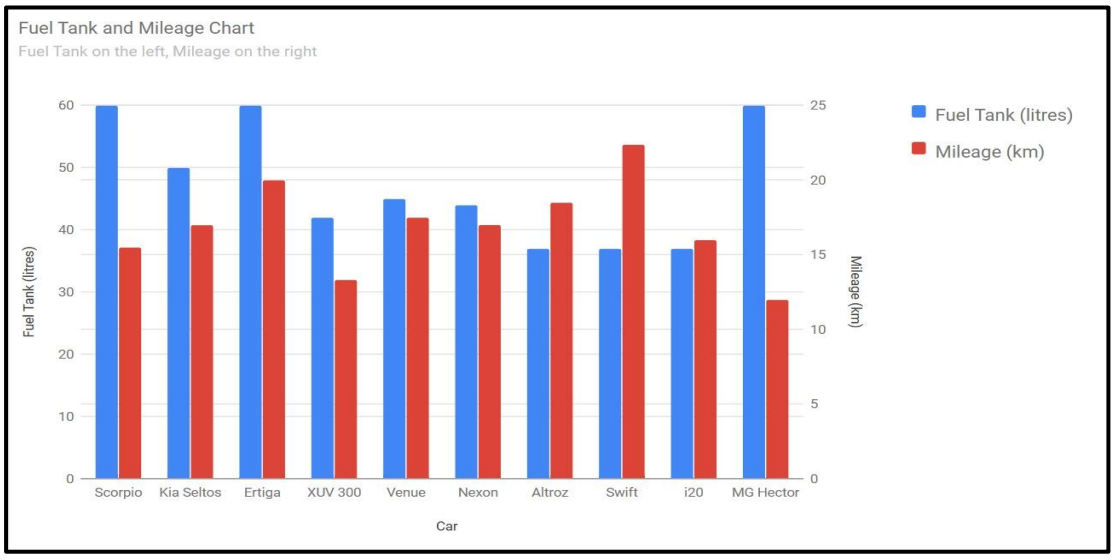

Fig 5.3: Car Fuel and Mileage Analysis

The image shows a bar chart comparing the fuel tank capacities and mileages of various car models. The chart has two y-axes: the left y-axis measures the fuel tank capacity in liters, represented by blue bars, while the right y-axis measures mileage in kilometers, represented by red bars. The car models are listed along the x-axis, including Scorpio, Kia Seltos, Ertiga, XUV 300, Venue, Nexon, Altroz, Swift, i20, and MG Hector. The chart visually compares each car's fuel tank size to its mileage. For example, the Scorpio has the largest fuel tank capacity but moderate mileage, while the Swift has a smaller fuel tank but high mileage. This chart helps quickly assess the trade-off between fuel tank size and mileage across different car models.

CHAPTER: 6
CONCLUSIONS & FUTURE SCOPE OF WORK

Chapter 6

.

CONCLUSIONS AND FUTURE SCOPE OF WORK

6.1 Web Scraping challenges

While implementing web scraper is not a difficult simple task, there are several issues that need to be addressed.

Firstly, traditional web scrapers can only target certain XPath or CSS selector in HTML markup but websites can have new features added anytime which cause changes to the entire website. Thus, web scraper developers need to update their code with new correct selectors making the cost for maintaining and monitoring web scraper higher than building one.

The crawler has to intelligently choose which XPath pattern to regulate its data grabbing behavior. This is similar to the use of CSS-selector in Ujwal et al. (2017) paper.

Secondly, as machine learning becoming ubiquitous, it is not unusual for websites to display personalized content. Web scraper cannot recognize which data is the correct data that it needs to retrieve. Many websites also check user IP Address to serve localized content. Web Scraper might extract content completely different from what normal user would see because their IP addresses are from different countries.

In summary, web scraper is a robust system that involves many technical components. The scraper needs to be able to understand markup language in the same way a browser does. It may also require that the scraper has its own browser engine in order to render the content correctly.

Sometimes the wanted content can also be in other text and binary format so the scraper needs special components to extract data from there. It is also important to plan on the data to collect and the pipeline needed to process it.

6.2 Future Scope of Work

1 Integration with other AI services: The AI web scraping service can be integrated with other AI services such as natural language processing, machine learning, and computer vision to provide more advanced features and capabilities.

2 Real-time data streaming: The service can be extended to support real-time data streaming, allowing users to extract data from dynamic web pages and receive updates in real-time.

3 Support for more data formats: The service can be extended to support more data formats such as XML, PDF, and images, making it more versatile and useful for a wider range of applications.

4 Improved error handling and data validation: The service can be improved to handle more complex error scenarios and provide more robust data validation, ensuring the accuracy and consistency of the extracted data.

5 Scalability and performance improvements: The service can be optimized to handle larger datasets and more complex web pages, improving its scalability and performance.

6 User interface improvements: The user interface can be improved to provide a more intuitive and user-friendly experience, making it easier for users to specify the data they want to extract and visualize the results.

7 Web Scraping Software: Nowadays several tools that can provide a custom web scraping. These programs can identify page data structure automatically or give a recording interface which eliminates the need for web scraping scripts. Furthermore, some of these software's can have a scripting function which can be used for extracting and transforming material, as well as database interfaces for scraping data and storing it in local databases

8 Computer vision-based web page analyzers: By visually scanning web pages like a person, computer vision and machine learning are being utilized to discover and retrieve important information.

REFERENCES

[1]Saurkar, Anand V., Kedar G. Pathare, and Shweta A. Gode. "An overview on web scraping techniques and tools." *International Journal on Future Revolution in Computer Science & Communication Engineering* 4.4 (2018): 363-367.

[2]Sirisuriya, De S. "A comparative study on web scraping." (2015).

[3]Banerjee, Ritu. "Website Scraping." *Happiest Minds Technologies* (2014).

[4]Almaqbali, Iqtibas Salim Hilal, et al. "Web scrapping: Data extraction from websites." *Journal of Student Research* (2019).

PUBLICATIONS

A paper following the research done on the topic has been sent for publishing to MAT Journals. The first page of the publication is attatched below.

Web-Sight: Automatic Web Scraping Tool

Prof. Jyoti Gupta
Assistant Professor
Department of Computer Science and Engineering
Bhilai Institute of Technology, Durg, India
Email: jyotigupta@bitdurg.ac.in

Rohit Kushwaha
Student
Department of Computer Science and Engineering
Bhilai Institute of Technology, Durg, India
Email: rkushwaha8610@gmail.com

Yaman Kumar Sahu
Student
Department of Computer Science and Engineering
Bhilai Institute of Technology, Durg, India
Email: yamankumarsahu.453@gmail.com

Vansh Dharmani
Student
Department of Computer Science and Engineering
Bhilai Institute of Technology, Durg, India
Email: vanshdharmani2907@gmail.com

Abstract

The project aims to develop an AI-powered web scraping service that enables users to extract data from websites in a fast and efficient manner. The service utilizes a combination of web scraping techniques, natural language processing, and machine learning algorithms to automatically identify and extract relevant data from web pages. The service provides a user-friendly interface that allows users to specify the URL and the data they want to extract, and then generates a structured dataset that can be easily analyzed and visualized. The service also includes features such as data validation, error handling, and data formatting to ensure the accuracy and consistency of the extracted data. The project is implemented using a combination of Python, JavaScript, and HTML/CSS, and is designed to be scalable, reliable, and secure. The service has the potential to revolutionize the way businesses and researchers collect and analyze data from the web, and has numerous applications in fields such as market research, competitive intelligence, and data journalism.

Keywords: *Web scraping, Extract, Data validation, Data formatting*

PLAGIARISM REPORT

Similarity Report

PAPER NAME

Web-Sight: Automatic Web Scraping Tool

WORD COUNT
7941 Words

CHARACTER COUNT
50450 Characters

PAGE COUNT
55 Pages

FILE SIZE
6.7MB

SUBMISSION DATE
May 10, 2024 2:34 PM GMT+5:30

REPORT DATE
May 10, 2024 2:35 PM GMT+5:30

14% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 11% Internet database
- 7% Publications database
- Crossref database
- Crossref Posted Content database
- 14% Submitted Works database

Excluded from Similarity Report

- Bibliographic material
- Quoted material
- Cited material
- Small Matches (Less then 10 words)

Summary