CPSC 4310

Advanced Data Processing

Programming Project Report

Web-Based Text Scraping for Consumer Product Data

Nora White

Overview

The focus of this programming project is to design and implement a web scraping system that access a websites that would ideally be used for price comparison purposes. Only the desired product data will be extracted based on some form of user input. This will avoid downloading of unnecessary pages which contributes to wasted time, data, and database resources, and also causes undue stress on the given website's server.

Strategy

Given the following scenario:

A user will access the GUI for the program and input a URL and a list of product brands and product names.

The program will request the robots.txt file, find the sitemap, and begin its search for the specific product based on the user's input parameters. Using Java's URL functions, the program will access the website files through a BufferedReader input stream. Using a combination of XPath and Regex expressions the files will be parsed for the expected values. The complete product will be put into a JSON file. The URL will also be stored to prevent having to search for the product's URL again. This JSON file can later be verified against a timestamp to see if any product data is out of date (such as a price drop). The product data will be displayed back on the GUI for the user to view.

Requirements

The following are a list of basic requirements that will be implemented for this project:

- GUI that allows a user to input Sephora's website URL, and a list of product brands and names to gather data on
- The program must be able to parse the website based on the robots.txt file, skipping over undesired URLs
- The program must be able to eliminate unwanted data from the product's page
- Useful product data will be saved into a JSON file
- Timestamp will be stored with JSON objects to check if product data has changed (such as price)
- GUI will display the results of the user's request
- Crawler will obey by the rules of the robots.txt file

Nice-to-Haves

The following are a list of features that would be beneficial to the program, but are not essential to its running:

- A user can input Shopper's Drug Mart's website URL, this would allow for price comparisons between products once the product data has been loaded into the database.
    - Shopper's Drug Mart's robots.txt does not provide an adequate sitemap, so the crawler will have to navigating the website to find the sitemap for beauty products
- Comparisons can be based on product size
- Program can be scheduled to run at recurring intervals of time

Constraints

The following are a list of constraints that need to be considered during implementation:

- Libraries and/or frameworks cannot make up the majority of the program.
    - If absolutely necessary, Selenium WebDriver or htmlUnit may be used for very small portions (but they will be avoided if/when possible)
- Time constraint of 4 weeks to implement

Programming Languages

The system will be developed in Java, with object storage handled with JSON.

Installation requirements

- Selenium webdriver: https://www.seleniumhq.org/download/
- Chromedriver: http://chromedriver.chromium.org/getting-started
- Netbeans: https://netbeans.org/
- Glassfish: https://javaee.github.io/glassfish/

How my Annotated Bibliography relates

The following is a general overview of how each annotated bibliography relates to this programming project.

K. Pol, N. Patil, S. Patankar and C. Das, "A Survey on Web Content Mining and Extraction of Structured and Semistructured Data," in 2008 First International Conference on Emerging Trends in Engineering and Technology, Nagpur, Maharashtra, India, 2008.

Based on this article, I can understand that the data that I was searching for is considered *unstructured data*, the XML documents are *structured data*, and the HTML pages are *semi-structured data*.

This article suggests creating a crawler to scour websites and classify their associated web pages based on their relevance to the search query. In my program, my crawler chooses the correct sitemap to use, and from that site map, discovers which URL to navigate to that matches the user's search query.

Further in, the authors suggest creating a DOM tree that contains a set of nodes. It appears to me that the nodes that they are referring to is the nodes that are accessible via xPath, which then creates a NodeList object that can be further queried. This NodeList is reduced to smaller structures when extracting data from the HTML page. This reduces the amount of time it takes to find the correct data, and prevents the scraper from traversing to places that are unnecessary and definitely do not contain useful data.

Outright, the authors of this article stress that it is much more difficult to mine content from semi-structured pages, which I can attest to.

W. Nadee and K. Prutsachainimmit, "Towards data extraction of dynamic content from JavaScript Web applications," in 2018 International Conference on Information Networking (ICOIN), Chiang Mai, Thailand, 2018.

In this article, the authors identified an important issue in data extraction from HTML pages: dynamically generated content. One of the biggest issues that I encountered was that the data I was receiving was missing entire sections because it hadn't been generated in time. By adding in Selenium and ChromeDriver as a non-headless web browser, I was able to overcome the issue of dynamically generated content.

The authors suggest extracting the data through DOM parsing, as mentioned in the previous article. I completed DOM parsing, but also used StAX as a faster alternative.

The authors make mention of a cached JSON approach to get around dynamically loaded web pages, but I don't believe that this is applicable to my design.

U. B. V. S, B. Gaind, A. Kundu, A. Holla and M. Rungta, "Classification-Based Adaptive Web Scraper," in 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), Cancun, Mexico, 2017.

I did not implement any strategies associated with this article other than finding one specific tag that was guaranteed to contain the data I needed, and beginning my search there to eliminate the chance of running into too much data that was unrelated to the task at hand.

S. Upadhyay, V. Pant, S. Bhasin and M. K. Pattanshetti, "Articulating the construction of a web scraper for massive data extraction," in 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, 2017.

This article deals mainly with implementation of crawlers and scrapers. They summarize issues mentioned in a previous article of this annotated bibliography related to dynamically loaded content, and constantly changing HTML document structures.

Most important in this article is the notion of ethics surrounding the traversal of the web page. It is important to practice caution and only scrape data that the site owner has delegated as safe for scraping. Based on how important being ethical about scraping is, I implemented the following features based on the robots.txt file:

- Determine the disallowed URLs. Based on these URLs, when the crawler is traversing the sitemap, I check the sitemap's URLs against the disallowed URLs to make sure I am not navigating to a website that is against the rules.
- Any time I request data from the server, I call a wait method that delays the process by the amount of time specified in the robots.txt crawl-delay.

B. Zhou, Y. Xiong and W. Liu, "Efficient Web Page Main Text Extraction towards Online News Analysis," in 2009 IEEE International Conference on e-Business Engineering, Macau, China, 2009.

DOM tree construction is a very time consuming process. The authors of this article proposed a solution that did not generate a DOM tree for this reason, and instead took the entire HTML page as one large string. Based on the word count of the inner-text of an HTML element, they extracted the data. I did not implement a strategy like this.

M. Marin, C. Castillo, A. Rodriguez and R. Baeza-Yates, "Scheduling algorithms for Web crawling," in WebMedia and LA-Web, 2004. Proceedings, Ribeirao Preto, Brazil, Brazil, 2004.

This article specifically talks about scheduling algorithms for web crawling. Had I implemented a comparison search of multiple websites, I would have looked more closely at this article. If I had implemented a second crawler that navigates to Shopper's Drug Mart's website, I would have generated multiple crawlers, that then generate multiple scrapers. In this way, the results would have been returned at the same time for both websites, resulting in more efficient crawling.

Evaluation Strategy

The following is the tentative structure of the evaluation strategy (some are pass/fail evaluations, others are evaluated to degrees):

1. Given user inputs, is the program able to navigate to a website, access the robots.txt file, and find the correct website URLs to download the product data from?
2. Is the program able to successfully extract product data and save it into the JSON?
3. Can the JSON have its data displayed on a GUI?
4. Can the gathered products have their data tested to make sure it is not out of date?
5. Length of time it takes the program to complete the task

The program will extract data from the website of Sephora (and if possible, Shoppers Drug Mart). The data that will be searched for is on luxury beauty products. Sephora's robots.txt file provides links to sitemaps that will be used to gather product links. Shoppers Drug Mart's robots.txt file does not provide an adequate sitemap for gathering product links for their beauty department so an alternative method of gathering the links would need to developed.

Evaluation Results

Given user inputs, is the program able to navigate to a website, access the robots.txt file, and find the correct website URLs to download the product data from?

Given user inputs, the program is able to navigate to a website, access the robots.txt file, and find the correct website URLs to download the product data from. This pass/fail requirement passes.

Is the program able to successfully extract product data and save it into the JSON?

Partially implemented. The program is able to successfully extract product data (see known issues) and display the data on the GUI, but the product data is not saved into the JSON. The image data was not possible to extract due to the XPath having difficulties using different namespaces. The image-url of a product was embedded in the attribute of an <image> tag, with the xlink:href attribute identifying it. I could not find a way to extract it. Saving the product data into the JSON was not implemented due to the amount of work and time required to simply get the data from the website. I can however, read the JSON objects in and create Product objects from them; but I used a library by Google called GSON to do this because I didn't want to do this manually.

Can the JSON have its data displayed on a GUI?

Based on an expected JSON structure, the JSON can have its data displayed on a GUI. The JSON is parsed into Objects and the objects are correctly displayed in an easy to read format.

Can the gathered products have their data tested to make sure it is not out of date?

Partially implemented. An initial JSON file has been created that houses 2 product JSON objects. These objects have a timestamp that is added to the created Product objects. With more time, I would have been able to implement a check to see if they were out of date, but as it became more and more apparent that parsing the HTML document was going to be very difficult, I ignored this functionality.

Length of time it takes the program to complete the task

To parse the robots.txt file, a simple InputStream and BufferedReader have been used to parse the lines of the file one at a time and determine if the line is useful.

To parse the sitemap.xml and the correct product xml file, two strategies have been implemented: DOM and StAX. (Please see post mortem as well)

DOM Method

The DOM method creates a tree of nodes to parse the file. This method is slow, especially for large files, since it reads in the entire XML file before parsing.

- The Crawler runs at a speed of around 26 seconds per correct URL pulled.
- The Scraper runs at a speed of around 26 seconds per product URL (shorter if the product URL is determined to have the incorrect brand).

StAX Method

StAX differs from DOM in that it uses a “pull” model. As it reads the XML file, the parser generates events that can be used to read the data more efficiently. The events used in this project are “START_ELEMENT”, “CHARACTERS”, and “END_ELEMENT”. START_ELEMENT signals that the beginning tag has been reached. CHARACTERS signals that data within the tags have been reached. END_ELEMENT signals that the end tag has been reached. StAX is faster than DOM because it reads the document from top to bottom without having to generate a tree of nodes.

- The Crawler runs at a speed of around 1.4 seconds per correct URL pulled.
- The Scraper was not implemented due to the amount of time and work required to complete the DOM Scraper.

Post Mortem

The following is a summary of learned experiences from completion of this project.

Web crawling/scraping is inherently slow

Due to the necessary time-constraint of the crawl-delay found in the robots.txt file, the entire process takes a minimum (but lengthy) amount time. Given a crawl-delay of 5 seconds:

Request	**Crawl delay**	**Minimum total time**
Robots.txt	5	= 5
Sitemap.xml of other sitemaps	5	= 10
n sitemap.xml (specifically the sitemap that contains every product URL)	5*n*	= 10+5*n*
m HTML files	(5*2)*m* Where 2 is the number of requests made per HTML page: 1 request to navigate to it with Selenium, 1 request to pull the HTML with JSoup	= 10 + 5*n* + (5*2)*m* = 10 + 5*n* + 10*m*

So out of the duration of the entire process, 10 + 5*n* + 10*m* is amount of time that is passing just from waiting the crawl delay between requests. There may be places where the crawl delay can be omitted, but for safe practice, any time there was a request made, I made my process wait immediately beforehand.

DOM parsing is slow

Parsing an XML or HTML file with a DOM tree is very slow. The standard test query that I would run was a brand of “benefit” and a product name of “porefessional”. The returned results are 7 products that match the query. The average time for the DOM crawler to find the correct URLs was around 26 seconds per found URL. Coincidentally, it takes the DOM parser around 26 seconds to crawl the HTML page. I quickly learned that it was unreasonable to progress further in the project without implementing a different method of crawling. The alternative I chose was StAX. StAX was able to bring the processing time of the crawler down from around 186 for 7 products seconds to 10 seconds for 7 products. Due to time constraints and multiple issues encountered with HTML parsing, a StAX crawler was not implemented.

For loops drastically increase processing time

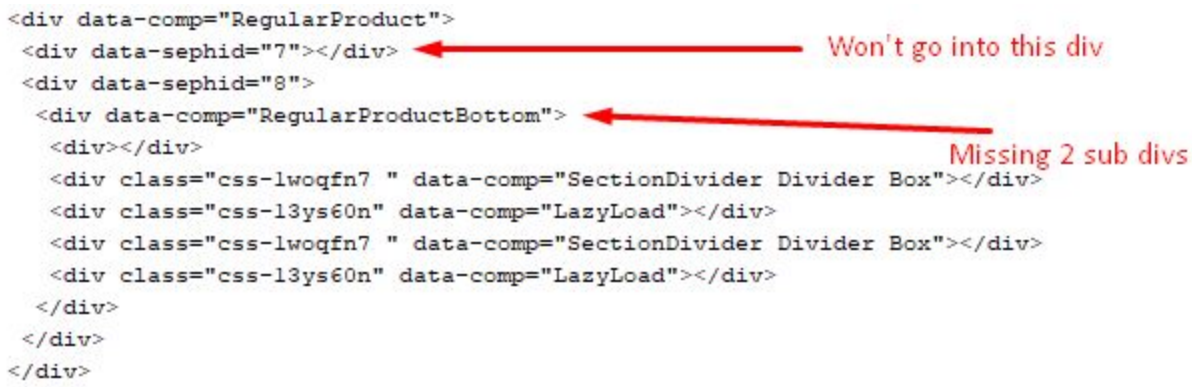
In the StAXCrawler, the length of time for the crawler to process the sitemap increases from around 10 seconds to around 136 seconds when I added in the checks for the disallowed URLs (found in robots.txt). To bring the speed back down to 10 seconds, I simply saved the size() of the ArrayList into an int and used that for faster checks rather than calling the size() function every time.

DOM and StAX are not designed for HTML

DOM and StAX are designed for XML, not HTML. HTML does not need to be well-formed and tags don't always need closing tags, so building a node tree from Sephora's website was impossible. For example, in the following HTML snippet, the <image> tag is self closing, but this does not prevent the page from being generated.

```
<div>
      <img src="..." />
</div>
```

Using an external library to help clean up the ill-formed HTML was a difficult process. I chose JSoup to clean the page since most sources online suggest that instead of manually parsing the HTML file, JSoup should be used instead. After running JSoup's "clean" method, I expected to have a well-formed document returned. However, the results of the clean were as follows (long lines cut off in screenshot to make code more readable):



So out of the entire document, the only data that I need is found in these two divs, but JSoup will not go deeper into them. I later determined that this was due to the page having only partially loaded by the time the input stream of the HTML page was sent to JSoup.

Selenium WebDriver does not work well with Glassfish

After learning that the website was not loading fully before I requested the input stream, I needed to find a way to wait for the page to load before requesting the content. I

implemented by system using JSPs and servlets. Using a skeleton from a project that I had done several years back, I used Glassfish as the server. Selenium WebDriver was a great solution, but it did not play well with Glassfish. After several hours of searching, I found one GitHub comment that suggested a change to the glassfish-web.xml file: change the class-loader delegate from true to false. Upon making this change, Selenium WebDriver integrated successfully with Glassfish.

After everything, this system only works with Sephora's website

It was difficult enough trying to get the data when I hardcoded the exact path to the data. It would require much more work (and give a much longer run time) to find the data with a relative path, or without knowing what the path was at all. This system would need to be completely re-done to accommodate different websites, their HTML design, and even just their URL structure.

Going forward, the things I would do differently and/or implement are…

The following are a list of things that I would do differently or implement in the future of this project.

- Compare the results of JSoup's parsing with StAX parsing.
- Find Shopper's Drug Mart's sitemap. There are two websites for shoppers, their regular website, and their beauty department website. After finding the robots.txt specifically for the beauty department, I was excited to see what their sitemap looked like. However, the sitemap that they provide *doesn't exist* and gives a 404 error if you try to navigate to it.
- Prevent a user from inputting the wrong website by either doing some concatenation of assumed URL substrings or doing it the long and slow way of navigating to google with Selenium and searching for the website with JSoup/whatever method of parsing an HTML page is found to be fastest.
- Implement the system where each query is a "job". This would allow the user to create a query and have it run in the background, but also add additional queries for processing immediately after one another. Ideally, after each job is complete, an email or notification of some sort would be generated. The user could then view their completed job.

- Learn more about how to navigate to the correct HTML tag without knowing the structure of the document. I was able to significantly cut down on the amount of HTML to parse in the DOM Scraper because I knew which div ID contained all of the useful data. By limiting my search to that, I can assume that the duration of time the DOM scraper was drastically reduced.

Overall, this was an aggressive project

I didn't expect this project to be as difficult as it was. I encountered many setbacks, most of the time regarding vague error messages. I also discovered that there is a fine line between what libraries are considered acceptable, and when using a library: how much can be used of it before it takes over the project.

One of the most unfortunate realities of building a web crawler/scraper that I found is looking for help when you have a question. Most of the time, the only answers to my questions on forums/blogs be: "You shouldn't attempt to do it yourself, you should use a library like JSoup because it's much easier." I would have liked to have used 0 libraries, but unfortunately it is necessary to use a resource like Selenium (or ideally, a headless browser), and due to time constraints, it is unrealistic to not use an existing library to assist you in some way, even if it is just with pre-processing.

The research papers that I studied all agreed that it was a large challenge to make a web scraper that works across multiple pages. Right off the bat I encountered the same issues that they described:

- The scraper needs to be remade when the structure of *any* documents change.
- The scraper can parse the entire page, but it is not efficient this way. Therefore it is necessary to limit the area in which the scraper can traverse, but it can be difficult to determine this without hardcoding the tag/node from which to begin its search.

Despite these issues, I'm very happy with my results. This is a project that I have been wanting to do for several years, but have never had the time or the drive to do it. I'm glad that I can say that I achieved most of the requirements that I set for myself, and have a good starting place for continuing to build this, post-graduation.

Known issues

The following is a list of known issues:

- JSoup does not process “ ” or similar escaped characters as expected. This results in the DocumentBuilderFactory not being able to create a Document off of the JSoup results. A search with the following input does not work due to this issue:
    - Brand: Benefit
    - Product: Happily

  There is only one product to search for: Benefit’s “Happily Ever Laughter Mini Set”. Somewhere in this document there is an “ ”. When trying to create a JSoup Document from this HTML file, it will not be created, therefore the product will not be created or displayed on the GUI.

- The search functionality does not handle multiple words. The best two working examples of how the system works is on the following two searches:
    - Brand: *Benefit*, Product: *Porefessional*
        - Returns 7 results successfully
    - Brand: *Dior*, Product: *Porefessional*
        - Returns 0 results successfully, because *Dior* does not make a product called *Porefessional*.

  However, if you were to put in “Porefessional mini”, there will be no results returned due to the lack-of logic put in for comparing the URL to the product.

  The only way to find the correct product in this system, is to go through the full list of all product URL’s in the product sitemap. So, if a URL only contains part of the full product name that the user has input, the result would hypothetically return with 0 results. But, if the user enters “porefessional mini”, it is being compared against a url that contains “porefessional-mini” with a dash.