Veridion Internship Project

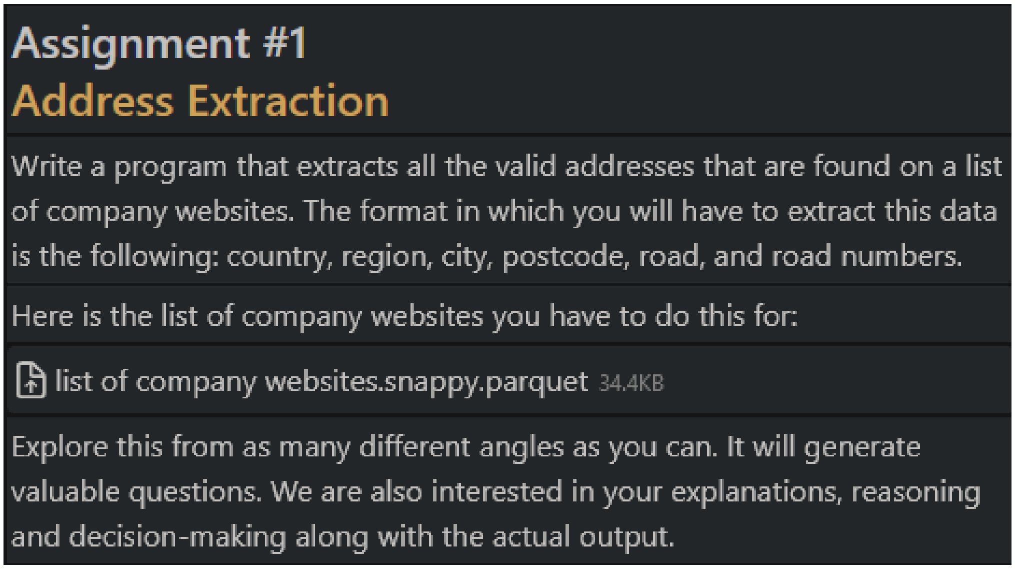
(Solution & Thought Process)

Contents

Introduction

This project was made for the role of “Deeptech Engineer” Intern.

Out of the 3 available tasks I have chosen the first one as it was, in my opinion, the most interesting and challenging of all. Also, I’ve worked on some web scraping projects by myself and I’m passionate about this topic.



Assignment #1

Address Extraction

Write a program that extracts all the valid addresses that are found on a list of company websites. The format in which you will have to extract this data is the following: country, region, city, postcode, road, and road numbers.

Here is the list of company websites you have to do this for:

list of company websites.snappy.parquet 34.4KB

Explore this from as many different angles as you can. It will generate valuable questions. We are also interested in your explanations, reasoning and decision-making along with the actual output.

Task for reference ^

Project Setup

- Clone the project from github https://github.com/Git-Lukyen/Veridion-Assignment.
- Add a Python interpreter to the project.
- Sync currently installed Python modules with “Requirements.txt”
- Run the “main.py” file found in the main directory

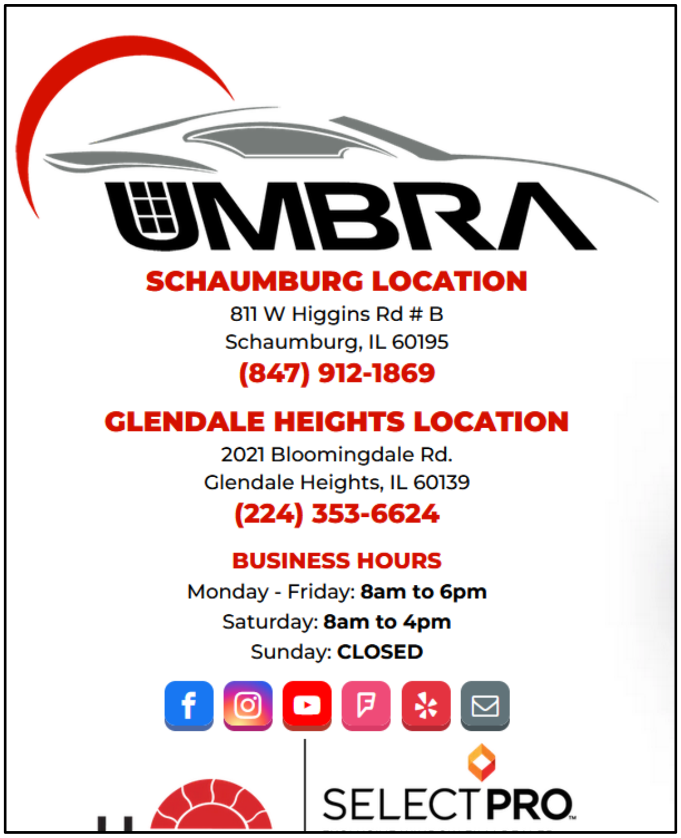
Input File

The first challenge I've encountered was actually extracting the links from the input file as I've never worked with ".parquet" files before. After a bit of research and trial & error I've implemented the solution of using the Python module "pandas" to create a DataFrame from the input file.

After this it's pretty straight forward, get the value from each entry in the DataFrame.

Analyzing the Data

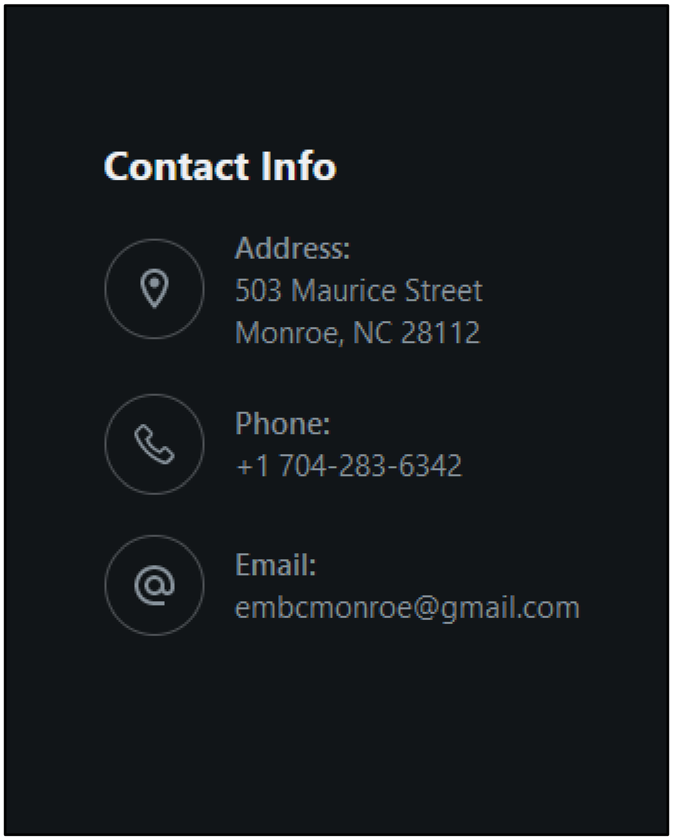
First step of web scraping is to analyse the web sites you have to work with. I chose the first 5 – 10 links, as well as other links at random and started looking at their structure. What exactly can I use from website to website that's similar enough and gives me the confidence that it's an address. Well let's see…



SCHAUMBURG LOCATION
811 W Higgins Rd # B
Schaumburg, IL 60195
(847) 912-1869

GLENDALE HEIGHTS LOCATION
2021 Bloomingdale Rd.
Glendale Heights, IL 60139
(224) 353-6624

BUSINESS HOURS
Monday - Friday: **8am to 6pm**
Saturday: **8am to 4pm**
Sunday: **CLOSED**



Contact Info

Address:
503 Maurice Street
Monroe, NC 28112

Phone:
+1 704-283-6342

Email:
embcmonroe@gmail.com

These are just 2 examples from the first 2 websites. But one thing is in common, the format of the address.

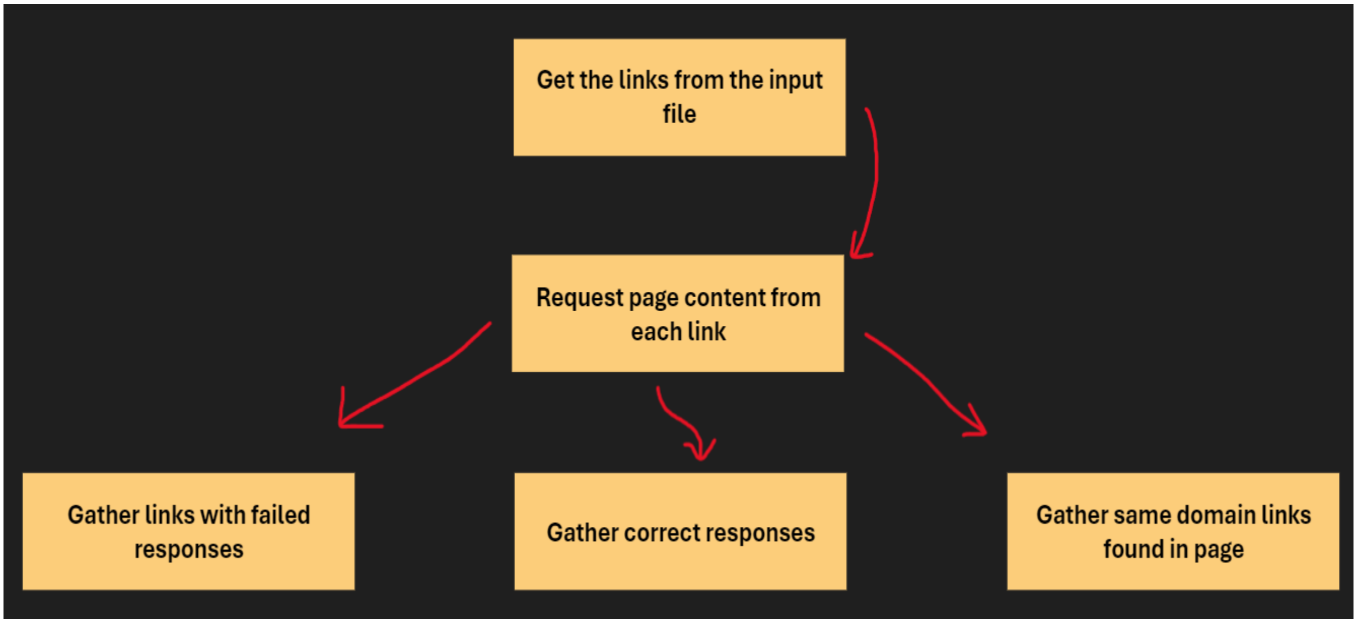
- zipcodes are formatted like so: “Region,” + “State/Country Code” + “Number”
- streets are formatted like so:”Number” + “Street Name” + one of many abbreviations (Ex: Rd. / Road, St. / Str. / Street, …)

Using this information we can actively search the pages for such formatted text.

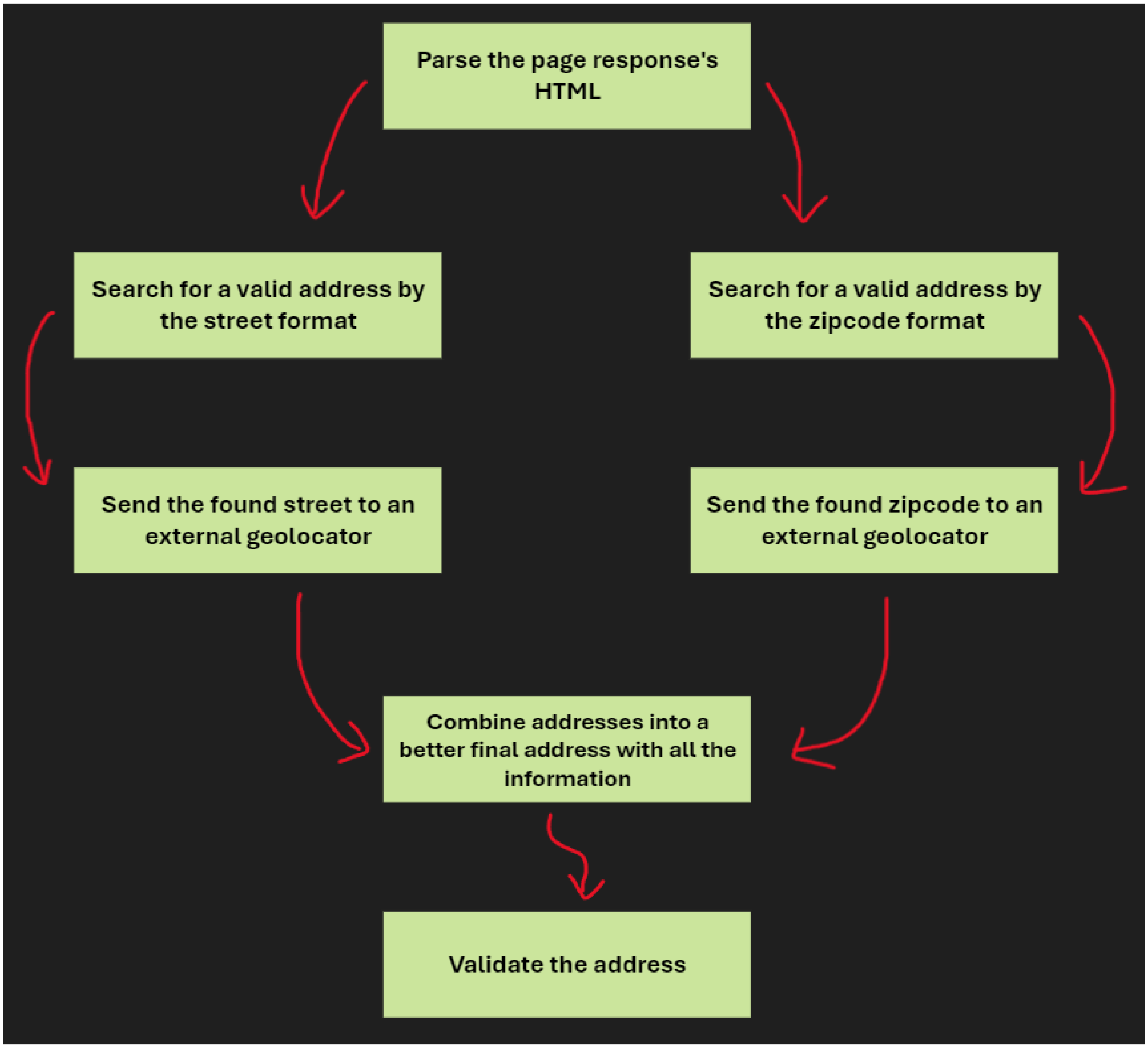
Solution Structure

Now that I had a possible solution in mind, it was time to lay out the structure of the project.

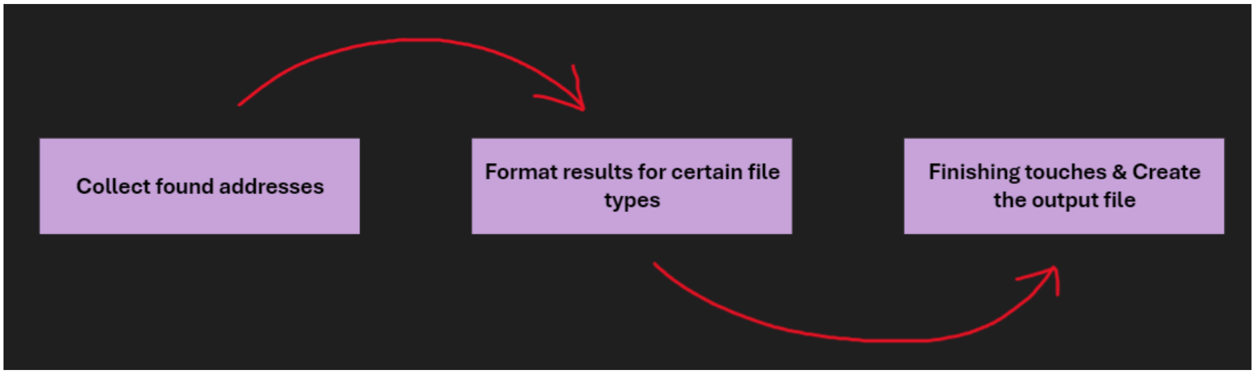
Input & Requests structure:


Get the links from the input file
Request page content from each link
Gather links with failed responses
Gather correct responses
Gather same domain links found in page

Address Extraction structure:


Parse the page response's HTML
Search for a valid address by the street format
Search for a valid address by the zipcode format
Send the found street to an external geolocator
Send the found zipcode to an external geolocator
Combine addresses into a better final address with all the information
Validate the address

Output structure:


Collect found addresses
Format results for certain file types
Finishing touches & Create the output file

Input

I first get the “links.snappy.parquet” file found in the “Input_Files” directory and create a DataFrame using “pandas” from it. Then, for every link inside of the DataFrame’s values I append it to “https://”, add it to a list and return that list.

Requests

To get a response from a page you need to make a request. Making requests sequentially isn’t that fast for ~2500 links so I’ve implemented an asynchronous approach.

I create a request task for each of the links and I group them into groups of 500. Then I send all the tasks at once for each group and gather all the responses.

Problems I’ve come across:

- Almost every page refuses a default request so a user_agent is needed.
- Too low of a timeout will lose many responses when sending asynchronous requests but this depends on the device hardware and internet connection.
- Some pages refuse requests from my location (Romania). I’ve tried implementing a proxy solution but working with free proxies is unreliable so I gave up on it. However, buying proxies and implementing them into my code would be easy so I just wanted to mention this is not an issue.

Failed Requests & Auxiliary Links

A large amount of the links given for the assignment are unavailable (either intentionally or not). If getting a response has failed I add the url to an array for future retries if specified.

Some websites might not have their address on the home page. To solve this issue I also get ~20 additional links with the same domain as the main url in case I can found the address there. Extracting more auxiliary links is possible, although I don't recommend it as it increases the execution time by a large margin.

Getting the Page Content

After we have the response we need a way to get objects from the page. I use a Python module called "Beautiful Soup" which is very popular in the web scraping scene. For the parsing engine I use "lxml" for it's fast speed and good results.

Using Beautiful Soup we can extract elements from the page using keywords or use other extraction methods.

Searching for the Address

Now that we have the page, how exactly do we get the address. I use regex as it's very fast for queries on strings and it allows you to do a bunch of neat tricks. The 2 patterns which I wrote and mainly use are:

- The zipcode pattern: "(^|\s)[a-zA-Z]{2} \d{4,6}(?:\s|$)". Brief explanation: It matches a 2 letter code followed by a number between 4 and 6 digits.
- The street pattern: "(?i)(^|\s)\d{2,7}\b\s+.{5,30}\b\s+(?:road|rd|way|street|st|str|avenue|ave|boulevard|blvd|lane|ln|drive|dr|terrace|ter|place|pl|court|ct)(?:\.|\s|$)" . Brief explanation: It matches a number that has between 2 and 7 digits, followed by a street name between 5 and 30 characters and then one of the common abbreviations.

This ensures a high probability of getting a correct address (backed up by personal testing) because it follows a specific formatting usually found on the websites.

After we have the zipcode and the street we use them as a query for an external geolocator. I use “Nominatim” as it is open-source but I think a better alternative would be Google Maps. I couldn’t get my hands on a Google Maps api key because for that I need to register my project and get an approval, which is unlikely in this time frame of a week.

This helps me get a more structured and correct address from the 2 pieces of text found on the page.

I then combine the 2 found addresses from each query into one final address that has one value for each field based on priority (for example the address found from the street has a higher priority for the street related fields than the zipcode one).

Finally, if the address has less than 4 fields with a not-null value then I consider it to be wrong or not accurate enough and I discard it.

This process is repeated for the retried links that give a valid response as well as for the auxiliary links. The final result is satisfactory and correct most of the time.

Creating the Output File

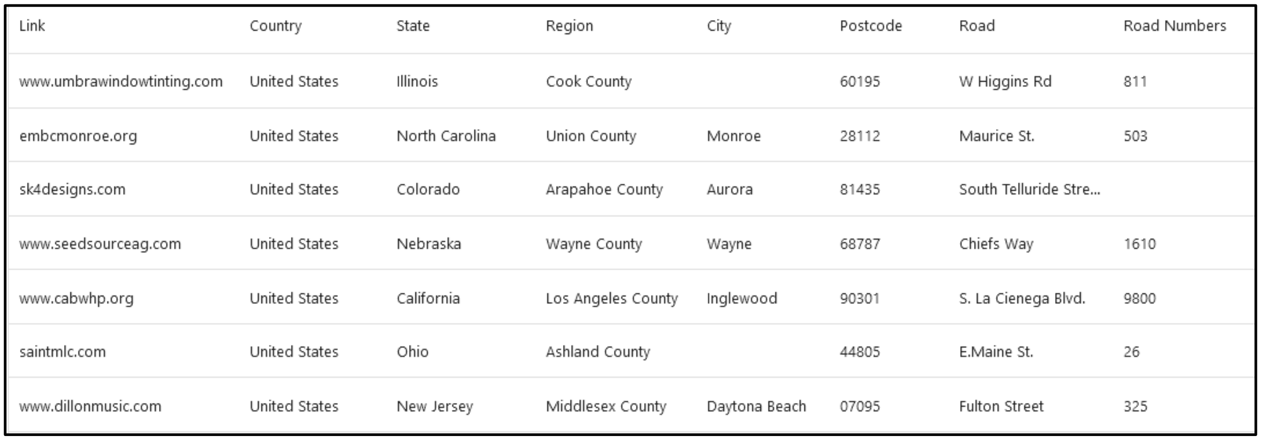
After the entire process is done, I gather all of the data into one place and create an output file with all of the found addresses. My program supports 2 output file types, that being json and .parquet, but more can be added.

Results

My program got 548 addresses or about 22% of the total websites with a total runtime of 76 minutes. Considering a big chunk of the links either don’t work or don’t have an available address I find this result satisfactory. The output files I got after running the program are available in the “Output_Files” directory so you don’t have to wait that long.

Links to output files: https://github.com/Git-Lukyen/Veridion-Assignment/tree/main/Output_Files

Preview:



Link	Country	State	Region	City	Postcode	Road	Road Numbers
www.umbrawindowtinting.com	United States	Illinois	Cook County		60195	W Higgins Rd	811
embcmonroe.org	United States	North Carolina	Union County	Monroe	28112	Maurice St.	503
sk4designs.com	United States	Colorado	Arapahoe County	Aurora	81435	South Telluride Stre...	
www.seedsourceag.com	United States	Nebraska	Wayne County	Wayne	68787	Chiefs Way	1610
www.cabwhp.org	United States	California	Los Angeles County	Inglewood	90301	S. La Cienega Blvd.	9800
saintmlc.com	United States	Ohio	Ashland County		44805	E.Maine St.	26
www.dillonmusic.com	United States	New Jersey	Middlesex County	Daytona Beach	07095	Fulton Street	325

Error Handling

Every good project has error handling, and so does mine. For everything that may raise an error even if it's not necessarely wrong (for example parsing the page content with Beautiful Soup or querying an address with a geolocator) my project has a "try except" statement and will print out an error to the console if anything happens so it doesn't stop the program entirely.

Going the Extra Mile

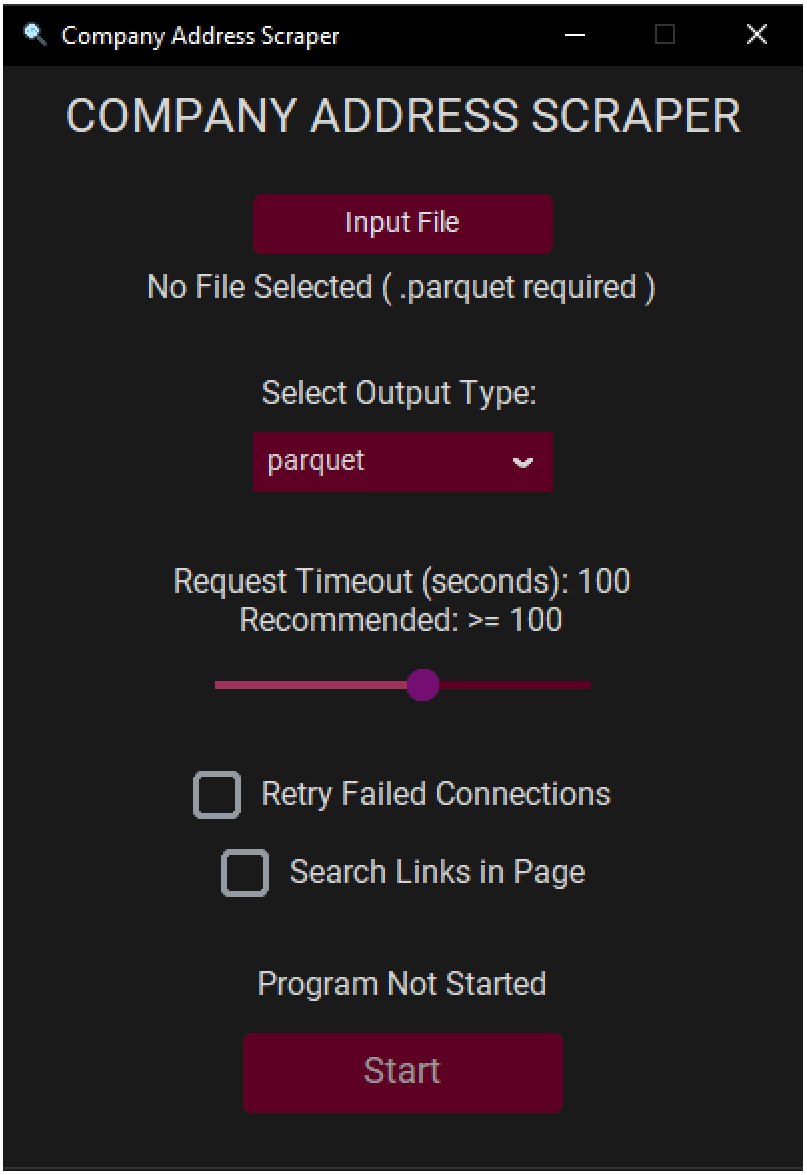
As a nice touch and something that might differentiate me from other projects is the User Interface I've made for the project. The interface makes it easy for a user to operate the program with custom parameters.

Functionalities:

- Select an input file path for ".parquet" files.

- Select the file type for the output file. Currently supported types: json & parquet.
- Select the request timeout value using a slider.
- A checkbox for whether to retry failed requests or not.
- A checkbox for whether to search for auxiliary links in the page.
- A status label that shows what the program currently does.
- A start button that starts the process.

Interface preview:



COMPANY ADDRESS SCRAPER

No File Selected (.parquet required)

Select Output Type:

parquet

Request Timeout (seconds): 100
Recommended: >= 100

□ Retry Failed Connections

□ Search Links in Page

Program Not Started

Final Notes

I’d like to say that I really liked the project and I find what Veridion does to be really interesting.

Also, if the “requirements.txt” file seems to have a lot of modules listed it is because I’ve had some issues with the “Sync Requirements” function in PyCharm not adding some of the modules required for the project and so I’ve added all the modules in my project but most of them come pre installed.