Web Crawling and Data Organization Report for CS104 Python project

Mayank Motwani
22B1052

Abstract

This is the documentation of a Web Crawler, written in Python. This has been done as a course project in the CS104(Software Systems Lab) course run in the 2022-2023 Spring Semester at IIT Bombay.

Contents

1 Introduction

This report provides an overview and analysis of the Python code used for web crawling and data organization for the CS104 Python Web Crawler Project. The code consists of two main scripts: *crawler.py* and *organizer.py*. The *crawler.py* script is responsible for crawling websites and extracting links, while the *organizer.py* script organizes the extracted data and generates statistics.

2 Implementation of the code

In what follows, I outline how the code works on a preliminary basis and the interrelations different files have with each other during the working. Detailed description will be provided in the respective sections for different files

- The code start by running crawler.py which assigns some global variables for output-file, threshold depth and visited links
- Main() is called subsequently. The code parses command line arguments with necessary error handling and stores them in respective variables
- Some files are created to assist in the process of crawling and the main crawling function get_page_links() is called.
- Get_page_links() is a recursive function to get links from a web page up to a specified depth. It takes required inputs in the form of url, depth, domain, output file and some subsidiary files
- Fetching of HTML content occurs with appropriate exception and redirection handling. A BeautifulSoup object is created for HTML parsing
- HTML code is crawled for the required src and href attributes along with required relative link handling and domain checking.
- The links are added to subsidiary files depending on whether they belong to the domain or not. Recursive crawling occurs until the threshold depth is reached or there are no unvisited links in the website's domain
- The crawling process ends and the files opened before are closed. The process of data organization starts as per the -o input i.e. to print to the output file or the command line
- The organizer.py (rather a function data_organizer of organizer.py) is called for organizing the crawled data. The required subsidiary files are opened in read mode and lines containing information about the links is read from them
- Data structures i.e. double dimensional are initialized to store the crawled information in a proper format and segregation on the basis of depth takes place. The basis of this process is the use of appropriate regex on the parsed data

- Common extensions such as html, css, js, etc. are organized into a list. Links are grouped on the basis of these extensions and organized depth wise. String operations and url-parsing are used in this process.
- After this, based on the required output format, either a file or the command line, the data is appropriately processed from these data structures.
- The process of data organization ends and files previously opened are closed. At this point, the data is completely organized as per the required format.

3 Customizations

Below, I list the three customizations I have done above the basic requirements of the code.

- The first customization is the addition of log statements at consequential points in the code to monitor if the code is correctly working.
- The second customization is the display of progress bars for loop functions, again to aid in the monitoring of the program in runtime. This is achieved by using a library in python known as 'tqdm' which is very easily integrated into any loop. The presence of a progress bar makes things seem nicer.
- The third customization involves making a pie chart for comparison of prevalence of different extensions in the crawled links. The pie chart represents the distribution of different extensions(i.e. the common extensions otherwise the chart becomes chaotic).

4 Working of the code : crawler.py Detail

4.1 Overview

The *crawler.py* script implements a web crawler that recursively visits links within a specified depth from an initial URL. It utilizes various libraries such as *requests*, $beautifulsoup4$, *urllib.parse*, and *time* for web scraping and processing. The $BeautifulSoup$ library is used to parse HTML content and the requests library to fetch web pages. The script extracts links from the 'src' and 'href' attributes of the HTML tags. The extracted links are then categorized based on whether they belong to the same domain or not.

4.2 Components and Functionality

Here is a rundown of the script's main components and their respective functionalities:

- Importing necessary modules:
  - *sys* - command line arguments
  - *requests* - for making HTTP requests
  - *BeautifulSoup* - for HTML parsing
  - *urlparse* from *urllib.parse* - for URL parsing
  - *re* - for regular expressions
  - *data_organizer*1 and *data_organizer*2 from the *organizer* module - for subsequent data organization
- Defining global variables:
  - *visited_links* : a set to keep track of visited links
  - *output_file* : a file object to write the output
  - *max_depth* : the maximum recursion depth for crawling
- Defining the *get_page_links* function:
  - This is a recursive function that takes a URL, depth, domain, output file, and two additional file objects as parameters and initiates the crawling process
  - It fetches the HTML content of the URL using the requests library.
  - It creates a BeautifulSoup object to parse the HTML content.
  - It crawls the web page for links in the "src" attribute of tags and adds them to the output file or the additional file depending on whether they belong to the same domain or not.
  - It crawls the web page for links in the "href" attribute of tags and follows the same process as above.
  - The function is called recursively with the extracted links to continue crawling to the specified depth.
  - The script writes the visited links to separate files: *recursion_domain_* and *recursion_non − domain_*.
- Defining the *main* function:
  - It parses command line arguments to retrieve the URL and threshold.
  - It sets up global file variables and opens the output and additional files for writing.
  - It validates the command line arguments.
  - It sets the maximum depth for crawling based on the threshold
  - It extracts the domain from the URL.
  - It starts crawling by calling the *get_page_links*() function with the initial URL, depth 0, domain, and file objects.

- After crawling, it closes the files and proceeds with data organization using the *data_organizer* functions based on whether an output file is provided.
- Finally, it closes the output and additional files.

4.3 Code Listing

```python
import sys
import requests
from bs4 import BeautifulSoup
from urllib.parse import urlparse, urljoin
import re
from organizer import data_organizer
from tqdm import tqdm
from time import sleep
# Global variables
visited_links = set()  # To keep track of visited links
output_file = None  # File to write the output
max_depth = 0  # Maximum recursion depth

def get_page_links(url, depth, domain, output_file, myfile1, myfile2):
    """
    Recursive function to get links from a web page up to a specified depth.
    """
    # Check if the maximum depth has been reached
    if depth >= max_depth:
        return
    # Fetch the HTML content of the URL
    try:
        response = requests.get(url)
        content = response.text
        # Handle redirection when response status code is 3xx
        if response.status_code in range(300, 400):
            redirected_url = response.headers.get('Location')
            if redirected_url:
                get_page_links(redirected_url, depth, domain,output_file, myfile1, myfile2)
    except requests.exceptions.RequestException:
        return
    # Create BeautifulSoup object for HTML parsing
    soup = BeautifulSoup(content, 'html.parser')
    # Crawling code for src attribute of tags
    for tag in tqdm(soup.find_all(src=True)):
        src = tag['src']
        # Resolve relative URLs
        src = urljoin(url, src)
        # Check if the link belongs to the same domain
        parsed_src = urlparse(src)
        if parsed_src.netloc == domain:
            # Add the link to the output file
            myfile1.write("recdep"+str(depth)+" "+"SRC"+" ")
            myfile1.write(src + '\n')
            # Check if the link has been visited before
            if src not in visited_links:
                visited_links.add(src)
                get_page_links(src, depth + 1, domain, output_file, myfile1, myfile2)
        else:
            myfile2.write("recdep"+str(depth)+" "+"SRC"+" ")
            myfile2.write(src + '\n')
        # sleep(0.001)
    # Crawling code for href attribute of tags
    for tag in tqdm(soup.find_all(href=True)):
        href = tag['href']
        # Resolve relative URLs
        href = urljoin(url, href)
        # Check if the link belongs to the same domain
        parsed_href = urlparse(href)
        if parsed_href.netloc == domain:
            # Add the link to the output file
            myfile1.write("recdep"+str(depth)+" "+"HREF"+" ")
            myfile1.write(href + '\n')
            # Check if the link has been visited before
            if href not in visited_links:
                visited_links.add(href)
                get_page_links(href, depth + 1, domain, output_file, myfile1, myfile2)
        else:
            myfile2.write("recdep"+str(depth)+" "+"HREF"+" ")
            myfile2.write(href + '\n')
        # sleep(0.001)

def main():
    # Parse command line arguments
    if len(sys.argv) < 3:
        print(
            "Usage: python3 web-crawler.py -u <url> -t <threshold> [-o <output-file>]")
        return
    # Setting up global file variables
    url = None
    threshold = None
    global output_file
    global myfile1
    global myfile2
    output_file = None
    myfile1 = None
    myfile2 = None
    # Read command line arguments
    for i in range(1, len(sys.argv)):
        if sys.argv[i] == '-u':
            url = sys.argv[i + 1]
        elif sys.argv[i] == '-t':
            threshold = int(sys.argv[i + 1])
        elif sys.argv[i] == '-o':
            output_file = open(sys.argv[i + 1], 'w')
    # Validate command line arguments
    if not url:
        print(
            "Mandatory url argument not provided, use tag -u to provide url to be crawled.")
        return
    if not threshold:
        print("Threshold is not provided and believe me you can't and wouldn't like to crawl
        return
    if threshold < 1:
        print("Please be reasonable and provide a threshold greater than equal to 1.")
        return
    print("___________________________________________________________________________")
    print('Command line arguments validated')
    sleep(1)
    global max_depth
    if threshold:
        max_depth = threshold
    else:
        # Setting a large value to max_depth to indicate crawling
        # until the end of the website if '-t' not provided
        max_depth = float('inf')
    # Parse the URL and extract the domain
    parsed_url = urlparse(url)
    domain = parsed_url.netloc
    # Open the files for writing
    name1 = "recursion_domain_"
    myfile1 = open(name1, "w")
    name2 = "recursion_non-domain_"
    myfile2 = open(name2, "w")
    print("___________________________________________________________________________")
    print('Crawling starts')
    sleep(1)
    # Start crawling
    visited_links.add(url)
    get_page_links(url, 0, domain, output_file, myfile1, myfile2)
    print("___________________________________________________________________________")
    print('Crawling ends')
    sleep(1)
    # Closing files in write mode to open them in read mode
    myfile1.close()
    myfile2.close()
    print("___________________________________________________________________________")
    print('Data organization starts')
    sleep(1)
    # Switching between organizing in the output file or the terminal according to the '-o'
    output = False
    if (output_file):
        output = True
    data_organizer(threshold, output_file, output)
    print("___________________________________________________________________________")
    print('Data organization ends')
    sleep(1)
    # Close the output file if it was opened
    if output_file:
        output_file.close()
    # Close the additional files
    if myfile1:
        myfile1.close()
    if myfile2:
        myfile2.close()

if __name__ == '__main__':
    main()
```

5 Working of the code : organizer.py Detail

5.1 Overview

The *organizer.py* script analyzes the data extracted by the *crawler.py* script and organizes it based on various criteria. It utilizes libraries such as *re*, *urllib.parse*, *tqdm*, and *matplotlib* for data processing and visualization. The code relies on several external libraries, such as *matplotlib*, *re*, *urllib.parse*, *tqdm*, and *time*

5.2 Components and Functionality

Here is a rundown of the components of the file organizer.py and relevant functionalities

- ***Import Statements***: The script imports several Python libraries/modules: *matplotlib.pyplot*, *re*, *urllib.parse.urlparse*, *tqdm*, *time.sleep*, and *matplotlib* for required data processing and visualization along with customizations for a better monitoring of the output. The matplotlib.use('TkAgg') line sets the backend for matplotlib to 'TkAgg', which is a specific backend for creating plots in a Tkinter GUI application.

- ***Function Definition***: The code defines a function named *data_organizer* that takes three parameters: threshold, *output_file*, and output.Here, note that the *output* parameter is a bool. The function is responsible for organizing and processing data from two input files: "recursion_domain_" and "recursion_non-domain_" .It separates the links into domain and non-domain categories and organizes them based on the depth. It performs various operations on the data and generates output either to a file (output_file) or the terminal based on the output parameter.

- ***Data Processing***:The function reads the content of the input files (recursion_domain_ and recursion_non-domain_) into separate lists (lines_domain and lines_non_domain). It then organizes the domain and non-domain links based on their depth (up to the provided threshold value) into 2-dimensional lists (domain_links and non_domain_links). The function further groups the links by their file extension, organizes them by depth, and stores the results in various data structures (depth_wise_links, extension_links).

- ***Output Generation***:Depending on the output parameter, the function generates either a file output or prints the information to the terminal. If output is true and an output_file is provided, the function writes the processed data to the file. If output is false, the function prints the processed data to the terminal.

- ***Pie Chart Generation***:The function generates a pie chart using the matplotlib library based on the collected data. It calculates the distribution of files for each file extension and displays the chart with labels and percentages.

- ***File and Resource Management***:The function closes the opened files (myfile1 and myfile2) at the end.

- Overall, the data_organizer function reads input files containing links, organizes and processes the data, generates output to a file or terminal, and creates a pie chart to visualize the distribution of file extensions.

5.3 Code Listing

```python
import matplotlib.pyplot as plt
import re
from urllib.parse import urlparse
from tqdm import tqdm
from time import sleep
import matplotlib
matplotlib.use('TkAgg')
# Data organizer for output file
def data_organizer(threshold, output_file, output):
    # Open the files for reading
    name1 = "recursion_domain_"
    myfile1 = open(name1, "r")
    name2 = "recursion_non-domain_"
    myfile2 = open(name2, "r")
    # Reading lines from the files
    lines_domain = myfile1.readlines()
    lines_non_domain = myfile2.readlines()
    # Initialize data structures to store links
    domain_links = [[] for _ in range(threshold)]
    non_domain_links = [[] for _ in range(threshold)]
    print("___________________________________________________________________________")
    print('Organizing domain files internally')
    sleep(1)
    # Segregating domain links according to depth
    for line in tqdm(lines_domain):
        for depth in range(threshold):
            search_string = "recdep"+str(depth)
            if re.search(search_string, line):
                link = re.search("[A-Z]+ \S*$", line)
                actual_link = link.group()
                actual_link = re.sub('^[A-Z]+ ', '', actual_link)
                domain_links[depth].append(actual_link)
        sleep(0.001)
    print("___________________________________________________________________________")
    print('Organizing non-domain files internally')
    sleep(1)
    # Segregating non-domain links according to depth
    for line in tqdm(lines_non_domain):
        for depth in range(threshold):
            search_string = "recdep"+str(depth)
            if re.search(search_string, line):
                link = re.search('[A-Z]+ \S*$', line)
                actual_link = link.group()
                actual_link = re.sub('^[A-Z]+ ', '', actual_link)
                non_domain_links[depth].append(actual_link)
        sleep(0.001)
    print("___________________________________________________________________________")
    print("""At this point, we have domain and non-domain links in a 2 dimensional list
    We have domain_links in a depth wise list and non_domain_links in a depth wise list""")
    sleep(1)
```

```
# Extensions upon which the data is grouped
extensions_common = ['html', 'css', 'js', 'png', 'jpg', 'jpeg', 'gif']
all_categories = extensions_common
print("___________________________________________________________________________")
print("""Grouping links by extension and organizing them depth-wise""")
sleep(1)
# Group links by extension and organize them depth-wise
depth_wise_links = [[] for _ in range(threshold)]
extension_links = {}
for depth, links in tqdm(enumerate(domain_links)):
    for link in links:
        parsed_link = urlparse(link)
        filename = parsed_link.path.split("/")[-1]
        extension = filename.split(".")[-1]
        name = extension+'_'+str(depth)
        if name not in extension_links:
            extension_links[name] = []
        extension_links[name].append(link)
        depth_wise_links[depth].append(link)
    sleep(0.001)
for depth, links in tqdm(enumerate(non_domain_links)):
    for link in links:
        parsed_link = urlparse(link)
        filename = parsed_link.path.split("/")[-1]
        extension = filename.split(".")[-1]
        name = extension+'_'+str(depth)
        if name not in extension_links:
            extension_links[name] = []
        extension_links[name].append(link)
        depth_wise_links[depth].append(link)
    sleep(0.001)
print("___________________________________________________________________________")
print("""Information to be written in output file is processed""")
sleep(1)
# List variables for drawing the pie chart
extension_for_graph = []
for extensions in extensions_common:
    extension_for_graph.append(0)

if output:
    # Complete processing of output-file
    if output_file:
        for level in tqdm(range(threshold)):
            s1 = f"At recursion depth {level}"
            output_file.write(s1 + '\n')
            total_files = len(depth_wise_links[level])
```

```
                s2 = f"Total links: {total_files}"
                output_file.write(s2 + '\n')
                for extensions in range(len(extensions_common)):
                    ext_count = sum(link.endswith(extensions_common[extensions]) for link i
                    extension_for_graph[extensions] = extension_for_graph[extensions] + int(
                other_links = []
                for extension_depth_s in extension_links.keys():
                    depth_indicator = int(extension_depth_s.split('_')[-1])
                    if depth_indicator == level:
                        extensions_without_depths = extension_depth_s.split('_')[0]
                        if extensions_without_depths in extensions_common:
                            s3 = extension_depth_s + ' : '
                            s4 = str(len(extension_links[extension_depth_s]))
                            output_file.write(s3 + s4 + '\n')
                            for link in extension_links[extension_depth_s]:
                                output_file.write(link + '\n')
                        else:
                            for link in extension_links[extension_depth_s]:
                                other_links.append(link)
                other_links_number = str(len(other_links))
                output_file.write("Miscellaneous"+' : ' + other_links_number+'\n')
                for link in other_links:
                    output_file.write(link + '\n')
                sleep(0.001)
    else:
        # Complete-processing of terminal output
        for level in tqdm(range(threshold)):
            s1 = f"At recursion depth {level}"
            print(s1)
            total_files = len(depth_wise_links[level])
            s2 = f"Total links: {total_files}"
            print(s2)
            for extensions in range(len(extensions_common)):
                ext_count = sum(link.endswith(extensions_common[extensions]) for link in dep
                extension_for_graph[extensions] = extension_for_graph[extensions] + int(ext_
            other_links = []
            for extension_depth_s in extension_links.keys():
                depth_indicator = int(extension_depth_s.split('_')[-1])
                if depth_indicator == level:
                    extensions_without_depths = extension_depth_s.split('_')[0]
                    if extensions_without_depths in extensions_common:
                        s3 = extension_depth_s + ' : '
                        s4 = str(len(extension_links[extension_depth_s]))
                        print(s3 + s4)
                        for link in extension_links[extension_depth_s]:
                            print(link)
                    else:
                        for link in extension_links[extension_depth_s]:
                            other_links.append(link)
            other_links_number = str(len(other_links))
            print("Miscellaneous"+' : ' + other_links_number)
            for link in other_links:
                print(link)
            sleep(0.001)
    #Pie-chart procedure
    file_extensions = all_categories
    files = [int(num) for num in extension_for_graph]

    cols = ['r','b','c','g', 'orange', 'lightskyblue', 'gold']
    plt.pie(files,
    labels=file_extensions,
    colors=cols,
    startangle=90,
    autopct='%1.1f%%')
    plt.title('Distribution of files found for a particular extension')
    plt.show()
    # Close the files
    myfile1.close()
    myfile2.close()
    if output_file:
        output_file.close()
```

6 Conclusion

The combination of the *crawler.py* and *organizer.py* scripts provides a comprehensive solution for web crawling, link extraction, and data organization. The *crawler.py* script efficiently crawls websites and extracts links, while the *organizer.py* script processes and analyzes the extracted data, generating useful statistics and visualizations.

While going through the process of creating this project, I took help of a few reference websites for different levels of detail, namely Beginner[1], Amateur[2] and Intermediate[3] and a book[4]. These articles were comprehensive and I used them mainly to learn about urllib and html parsing using beautiful soup.

References

[1] URL: https://www.zenrows.com/blog/mastering-web-scraping-in-python-crawling-from-scratch#one-url-at-a-time,-sequential.

[2] URL: https://www.pluralsight.com/guides/web-scraping-with-beautiful-soup.

[3] URL: `http://www.gregreda.com/2013/03/03/web-scraping-101-with-python/`.

[4] Ryan Mitchell. *Web Scraping with Python.* O'Reilly.