



```
23 out_file.write('\n')
```

**Listing 1:** Extract the URIs from the .JSON file

*Step 3: Extract links from the collected tweets*

The following python code will extract all the URIs from the .JSONL file and save the output in a output.txt file.

```
1 import json
2 import re
3
4 # Regular expression to match URLs
5 url_pattern = re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@
    .&+]|[*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
6
7 # Open the JSONL file containing tweets
8 with open('tweets-1545.jsonl', 'r', encoding='utf-8') as f:
9     # Loop over each line in the file
10    for line in f:
11        # Load the JSON data from the line
12        tweet_data = json.loads(line)
13        # Extract links from the tweet's text
14        links = re.findall(url_pattern, tweet_data['text'])
15        # If links were found, write them to the output file
16        if links:
17            with open('output.txt', 'a', encoding='utf-8') as out_file:
18                out_file.write('Tweet: {}\n'.format(tweet_data['text']))
19
20            out_file.write('Links:\n')
21            for link in links:
22                out_file.write('{}\n'.format(link))
23            out_file.write('\n')
```

**Listing 2:** Extract the URIs from the .JSONL file

*Step 4: Resolve URIs*

The following Python code will resolve the output.txt file and extract the links from that file and store it into separate resolved-urls.txt file

```
1 import requests
2
3 # read the URLs from the file
4 with open('output.txt', 'r', encoding='utf-8') as f:
5     urls = [line.strip() for line in f.readlines()]
6
7 # resolve each URL and save the final target URI in a new file
8 with open('resolved_urls.txt', 'w') as f:
```

```
9     for url in urls:
10         try:
11             response = requests.get(url, allow_redirects=True, timeout
=5)
12             final_url = response.url
13             f.write(final_url + '\n')
14         except:
15             print('Error resolving URL:', url)
```

**Listing 3:** Resolve the URIs to extract URLs from output.txt

*Step 5: Resolve URIs to the Final target URI and storing the unique URIs to file*

We will be collecting many shortened links (e.g., dlvr.it, bit.ly, buff.ly, etc.), and our goal is to obtain the actual HTTP 200 URI. To accomplish this, I wrote a Python code that redirects the shortened links to their final URI and saves the resolved links in a file. Additionally, we are performing exception handling to eliminate unwanted URIs. The output of the code is a file named "unique\_urls.txt" that contains all the unique links that have been resolved.

```
1 unique_uris = set()
2
3 with open('resolved_urls.txt', 'r', encoding='utf-8') as f:
4     for line in f:
5         uri = line.strip()
6         if uri not in unique_uris:
7             unique_uris.add(uri)
8
9 with open('unique_urls.txt', 'w') as f:
10     for uri in unique_uris:
11         f.write(uri + '\n')
```

**Listing 4:** Resolve URIs to the unique target URI

*Step 6: Remove the utm data from the links*

I used a script that extracts the base URL of each link and eliminates any parameters beginning with "utm" to remove the UTM data from the links. This was necessary because I encountered UTM errors when running the script. The output is a list of cleaned URIs, saved in a file named "cleaned-urls.txt".

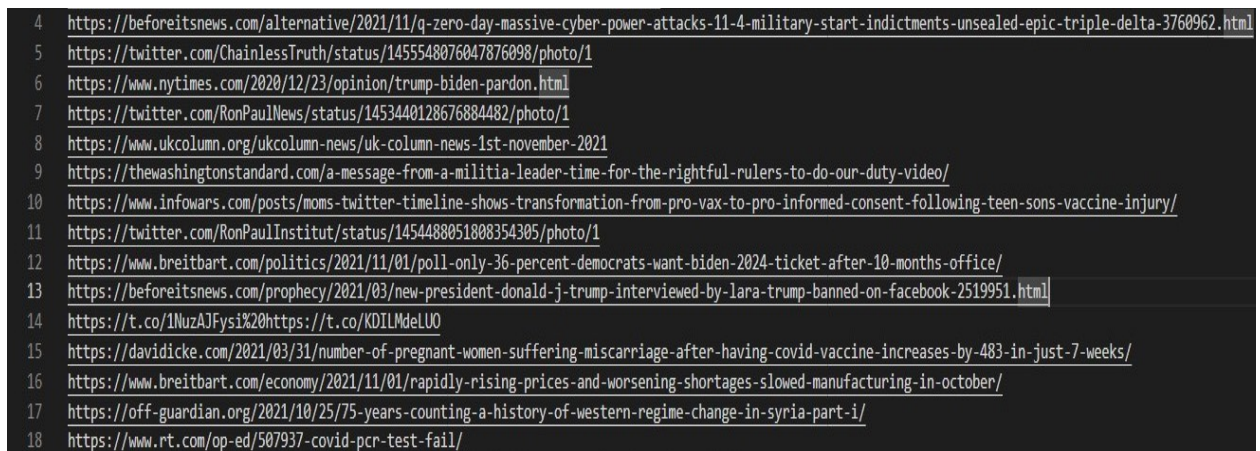
```
1 from urllib.parse import urlparse, parse_qs, urlencode, urlunparse
2
3 def remove_utm_params(url):
4     parsed_url = urlparse(url)
5     query_params = dict()
6     for key, value in parse_qs(parsed_url.query).items():
7         if not key.startswith('utm_'):
8             query_params[key] = value
9     new_query_string = urlencode(query_params, doseq=True)
```

```
10 new_parsed_url = parsed_url._replace(query=new_query_string)
11 new_url = urlunparse(new_parsed_url)
12 return new_url
13
14 input_file = open('unique_urls.txt', 'r')
15 output_file = open('cleaned_urls.txt', 'w')
16
17 for line in input_file:
18     url = line.strip()
19     new_url = remove_utm_params(url)
20     output_file.write(new_url + '\n')
21
22 input_file.close()
23 output_file.close()
```

**Listing 5:** Remove utm to resolve URIs to the final target URI

## Discussion:

Figure 1 shows the collected tweets in cleaned-urls.txt file.



**Figure 1:** collection of tweets

To obtain 1000 unique links, we typically need to collect more than 1000 tweets by accessing the Twitter API through a developer account and using the twarc library. However, for this project, the professor has already provided us with the required tweets in the form of JSON and JSONL files, so we don't need to access the Twitter API. As described in step 2 and step 3, I wrote Python code to extract only the tweet URIs while removing any unwanted key-value pairs. However, the extracted URIs included a significant number of shortened links that were not the original links. Therefore, in step 4, I developed Python code to replace all shortened links with their corresponding original links while also handling exceptions for unresolved links and discarding them accordingly.

Finally, the resolved URIs is stored to the file named "cleaned-urls.txt" which consists of more than 1000 unique URIs of tweets.

## Q2. Get TimeMaps for Each URI

### Answer

I have written the following Python program to obtain the TimeMaps for each of the unique URIs that were generated in Q1 of this report.

```
1 import sys
2 import requests
3 import os
4
5
6 if not os.path.exists("timemaps"):
7     os.mkdir("timemaps")
8
9
10 memento_analysis = open("memento_analysis.txt", "w")
11
12 with open("cleaned-urls.txt", "r", encoding='utf-8') as fobj:
13     count = 0
14     for url in fobj:
15         # fname = url.split("/")[2]
16         #if not os.path.exists(os.path.join("timemaps")):
17             #with open(os.path.join("timemaps", fname), "w") as
18         output_file:
19             try:
20                 response = requests.head("http://localhost:1208/timemap/
21                 json/" + url.strip())
22                 print(f'Writing URL: {url} : {response.status_code}')
23                 if response.status_code == 200:
24                     print(response.headers)
25                     if "x-memento-count" in response.headers:
26                         print(response.headers["x-memento-count"])
27                         memento_analysis.write(f'url.strip() : {response.
28                         headers["x-memento-count"]}\n')
29                     #mems = response.content.decode("utf-8")
30                     #new_timeMaps = mems[2:]
31                     #output_file.write(mems)
32                     #output_file.flush()
33                     count += 1
34                 print("URLs complete: " + str(count))
35             except requests.exceptions.RequestException as e:
```

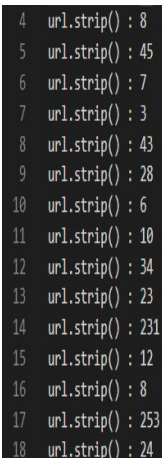
```
33         print("Exception: {} for link: {}".format(str(e), url.strip  
           ()))
```

**Listing 6:** Getting TimeMap for each URIs

## Discussion:

To obtain TimeMaps for the URIs generated in Q1, we are utilizing Memgator, which is a Memento Aggregator tool. We have extracted each URI from the final links file using a code, and all the final URIs are stored in a file called "cleaned-urls.txt". This file is used to send each URI to the ODU Memento Aggregator, which helps us to obtain the corresponding TimeMaps. Once we obtain the TimeMaps, we save each of them in a file named "memgator-analysis.txt". The output, which includes all the TimeMaps that we aggregated using Memgator in a txt format, is displayed in Figure 2.

Figure 2 shows the collection of TimeMaps.



```
4 url.strip() : 8
5 url.strip() : 45
6 url.strip() : 7
7 url.strip() : 3
8 url.strip() : 43
9 url.strip() : 28
10 url.strip() : 6
11 url.strip() : 10
12 url.strip() : 34
13 url.strip() : 23
14 url.strip() : 231
15 url.strip() : 12
16 url.strip() : 8
17 url.strip() : 253
18 url.strip() : 24
```

**Figure 2:** collection of tweets

## Q3 Analyze Mementos Per URI-R

In the second section of this report, we created separate TimeMaps and stored them in a file named "memento analysis.txt". For URIs with more than zero mementos, we must now determine the memento's count. To do so, I have developed the following code.

```
1 with open('memento_analysis.txt') as file:
2     data = file.readlines()
3
4 # initialize counts for each interval to 0
5 count_0 = 0
```

```
6 count_1_10 = 0
7 count_11_20 = 0
8 count_21_30 = 0
9 count_31_40 = 0
10 count_41_50 = 0
11 count_51_100 = 0
12 count_101_500 = 0
13 count_501_1000 = 0
14 count_1000_plus = 0
15
16 # iterate through the list and count numbers in each interval
17 for number in data:
18     number = int(number.strip()) # remove any whitespace and convert
    to integer
19     if number == 0:
20         count_0 += 1
21     elif number >= 1 and number <= 10:
22         count_1_10 += 1
23     elif number >= 11 and number <= 20:
24         count_11_20 += 1
25     elif number >= 21 and number <= 30:
26         count_21_30 += 1
27     elif number >= 31 and number <= 40:
28         count_31_40 += 1
29     elif number >= 41 and number <= 50:
30         count_41_50 += 1
31     elif number >= 51 and number <= 100:
32         count_51_100 += 1
33     elif number >= 101 and number <= 500:
34         count_101_500 += 1
35     elif number >= 501 and number <= 1000:
36         count_501_1000 += 1
37     elif number > 1000:
38         count_1000_plus += 1
39
40 # print the counts
41 print("Count of numbers in the list:")
42 print(f"0: {count_0}")
43 print(f"1-10: {count_1_10}")
44 print(f"11-20: {count_11_20}")
45 print(f"21-30: {count_21_30}")
46 print(f"31-40: {count_31_40}")
47 print(f"41-50: {count_41_50}")
48 print(f"51-100: {count_51_100}")
49 print(f"101-500: {count_101_500}")
50 print(f"501-1000: {count_501_1000}")
```

```
51 print(f"1000+: {count_1000_plus}")
```

**Listing 7:** Calculate the mementos by number

Mementos	URI-Rs
0-10	538
11-20	198
21-30	114
31-40	86
41-50	28
51-100	63
101-500	88
501-1000	6
Greater than 1000	32

## Discussion

This is a Python code that reads a list of numbers from a file called 'memento-analysis.txt' and counts how many numbers are in each interval. The code initializes counters for each interval to 0 and iterates through the list of numbers, incrementing the corresponding counter for each interval that the number falls into. Finally, the code prints out the counts for each interval.

## Q5 Explore Conifer and ReplayWeb.Page to archive the Web pages

### Answer

I opted for the topic of Hyperledger Fabric, a Blockchain framework, to develop a decentralized tool based on private Blockchain. I selected this topic due to my previous years of exploration in the field of Blockchain, where I observed how each release of this framework brings changes in the implementation process of older versions. I encountered no difficulties while archiving the web pages related to this topic, as all the archived pages appeared nearly identical to the original ones. A total of 764 URLs have been archived, with 12 pages archived in their actual page format. The remaining archived URLs are in the format of their implementation mode, such as image files, JSON files, HTML files, and other extensions. I didn't expected the Images count will be the highest among all others file-types.

Browser link where the WARC file is being located: <https://replayweb.page/?source=file%3A%2F%2Fhw2-20230228054045.warc#view=resources&urlSearchType=>



prefix

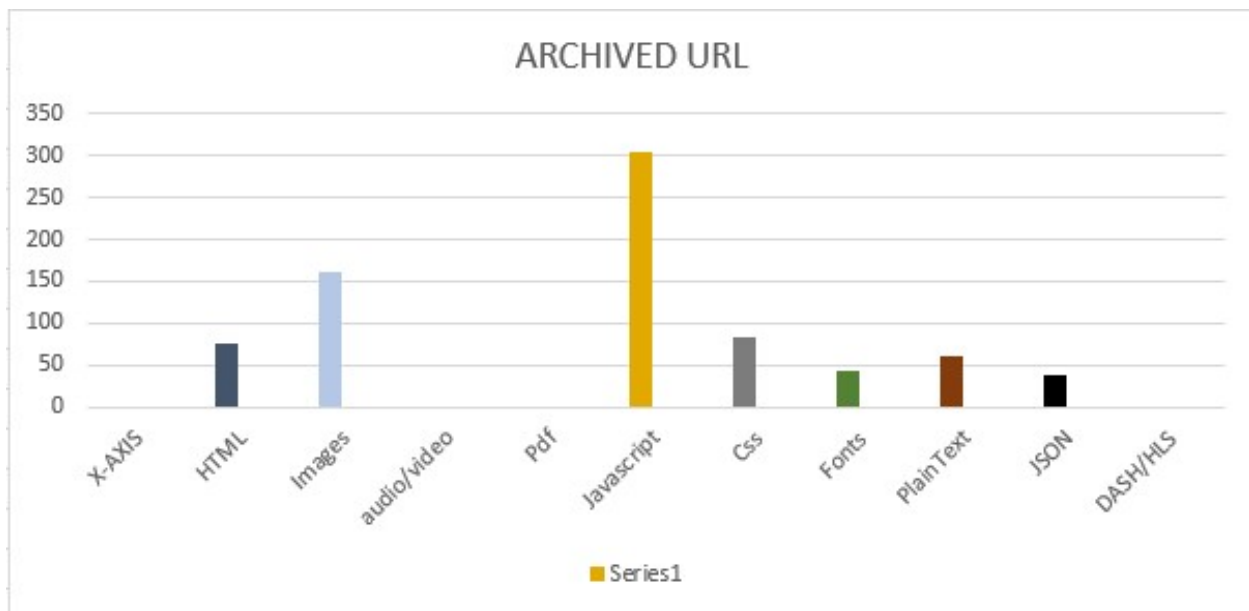
Figure 3 shows the number of pages archived using WARC file.

URL	Date	Mime Type	Status
http://c.amazon-adsystem.com/aax2/apstag.js	2/28/2023, 12:39:14 AM	application/javascript	200
http://c.amazon-adsystem.com/bao-csm/aps-commn/aps_csm.js	2/28/2023, 12:39:16 AM	application/javascript	200
http://cdnjs.cloudflare.com/ajax/libs/require.js/2.1.14/require.min.js	2/28/2023, 12:39:12 AM	application/javascript	200
http://js.hs-scripts.com/8112310.js	2/28/2023, 12:37:07 AM	application/javascript	200
http://m.servedby-buysellads.com/monetization.js	2/28/2023, 12:39:13 AM	application/javascript	200
http://tracking.oak-tree.tech/matomo.js	2/28/2023, 12:40:13 AM	application/javascript	200
https://1.www.s81c.com/common/carbon-for-ibm-dotcom/version/v1.23.0/202.js	2/28/2023, 12:37:35 AM	application/javascript	200
https://1.www.s81c.com/common/carbon-for-ibm-dotcom/version/v1.23.0/_commonjsHelpers.js	2/28/2023, 12:37:35 AM	application/javascript	200
https://1.www.s81c.com/common/carbon-for-ibm-dotcom/version/v1.23.0/button-cta.min.js	2/28/2023, 12:37:35 AM	application/javascript	200
https://1.www.s81c.com/common/carbon-for-ibm-dotcom/version/v1.23.0/button-group-item.js	2/28/2023, 12:37:35 AM	application/javascript	200

**Figure 3:** Number of pages archived using WARC file

Figure 4 shows the Bar chart created which describes the count of each file type which is been archived.

A total of 764 URLs have been archived, with 12 pages archived in their actual page format.



**Figure 4:** count of each file type being archived

The remaining archived URLs are in the format of their implementation mode, such as image files, JSON files, HTML files, and other extensions. I didn't expected the Images count will be the highest among all others file-types.

## References

*Following are the references which I used while doing my HW2, you can refer to some of links from where I took help.*

- MemGator, <https://github.com/oduwsdl/MemGator>
- TimeMaps, <http://www.mementoweb.org/guide/quick-intro/>
- Conifer, <https://conifer.rhizome.org/>
- Python requests, <https://docs.python-requests.org/en/master/>