HW2 CS432

Ondra Torkilson

October 11, 2020

Question 1

Pictured on page 2 is my python script for extracting the URIs, determining how many URIs I had to collect to get 1000 unique, how many of the original URIs were shortened, and how many of the unique URIs were initially redirected.

To get the URIs, I used Tweepy. I had to generate consumer keys in addition to my access keys for the TwitterAPI. In order to make sure I collected 1000 unique URIs, I made my exit condition test the length of a set constructed using the original list. As soon as the length equaled 1000, I moved on to requesting all of the 1000 unique URIs in order to determine if redirects occur. I experienced some connection exceptions when I finally moved up to collecting the full 1000 which explains the exception handling lines 36-37. To determine redirects, I checked the history attribute of the request response object which is returned as a list. An history list with length greater than 0 indicates redirection. To keep track, I placed the redirected URIs into their own set and tested the length of that set.

```python
import tweepy
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
import json
import requests
from re import search
import urllib3

# authorization tokens
consumer_key = "DopegZYSSV3dceUNrSG8eA1xO"
consumer_secret = "ul6RHlM25wrTiWuLVULylmDidsBWdBC1LZSD1ptWxcPGMEZLtH"
access_key = "233049998-avBODMjHqvWSgymtOaaNNE006A0Jm4bhnbYCo9NZ"
access_secret = "dFZo7rECe16kGs46ILHzrw0ppGm41PhRGfBFlp0eGFJKz"

originalList = []
shortenedSet = set()
redirectedSet = set()
uniqueSet = set()
uniqueList = []

class StdOutListener(StreamListener):

  def on_data(self, data):

    decoded = json.loads(data)

    uniqueSet = set(originalList)
    if len(uniqueSet) ==  1000:
      print(len(originalList))
      print(len(shortenedSet))
      uniqueList = list(uniqueSet)
      for url in uniqueSet:
        try:
          response = requests.get(url, timeout=30.0)
        except requests.exceptions.RequestException as e:
          pass
        if (len(response.history) != 0):
          redirectedSet.add(url)
        else:
          continue
      with open('json.txt', 'w') as outfile:
        json.dump(uniqueList, outfile, indent=2)
      print(len(redirectedSet))
      import sys
      sys.exit()

    else:
      try:
        for url in decoded["entities"]["urls"]:
          string = str(url)
          if search('twitter', string):
            continue
          else:
            originalList.append(url["expanded_url"])
            if url["url"] != url["expanded_url"]:
              shortenedSet.add(url["url"])
            else:
              continue
            #print (url["expanded_url"] + "\r\n")
            #print (url["url"] + "\r\n\n")
      except KeyError:
        print(decoded.keys())
  def on_error(self, status):
    print(status)

if __name__ == '__main__':
  l = StdOutListener()
  auth = OAuthHandler(consumer_key, consumer_secret)
  auth.set_access_token(access_key, access_secret)
  stream = Stream(auth, l)
  stream.filter(track=['YouTube'])
```

Figure 1: extractURIs.py

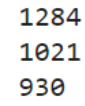

Figure 2: Original List Length, Shortened, Redirects

The image above are the results of the extractor. I collected 1284 URIs in total before I was able to get 1000 unique. 1021 of the original 1284 were shortened, which I test for on lines 56-59. I compare the main URL to the extended. Most all links shared through tweets are shortened to since Tweets are length restricted. The bottom number refers to how many of the unique 1000 links were redirects, and it came out to 930. That code, discussed previously, is on lines 33-41.

I dumped the unique URIs into a JSON file, with each URI separated by a new line. I had to do a bit of a hack here, since set do not contain serializable objects, and I had to make a list of the unique set so that I could output it as a string to the JSON file. This process can be seen on line 32 and lines 42-44.

Let's talk about lines 48 to the end. As long as the unique set did not have 1000 items, we are in that else statement which continuously pulls URIs and their attributes from the stream. To make sure I am not grabbing links to twitter, I test each link for the string 'twitter' before placing into the original list of URIs.

Lastly, I used the filter 'YouTube' while streaming, as this populated very quickly.

Questions 2 and 3

For question 2, I wrote a python script (Figure 3) to request timemaps from MemGator in JSON format (line 12.) For each timemap returned, I appended it to a text file (lines 19-20) so long as its not empty. I split my unique links into 10 batches, running the jsonGetTimeMaps script for 100 links at a time to protect against wasted time in case the script stopped running. I did end up combining batches 9 and 10. I needed sleep, and I knew that my other batches had been successful. After collecting all of the timemaps, I combined the resulting text files into a single text file of JSON format (I just had to add sqaure brackets to the beginning and end of the document and remove the last trailing comma. Then, I started analyzing the timemaps with a new python script, Figure 4.

The script in Figure 4 did a lot. It loaded the JSON data to my variable timeMaps (line 9). The first for loop broke the data for each timemap down into its respective parts using the JSON keywords. I extracted the original URI, the

```python
import subprocess
import json

with open('links9and10.txt') as json_file:
    data = json.load(json_file)
#print(data)

timeMaps = []
i = 0

for url in data:
    cmd = "memgator-windows-amd64.exe " +"--format=JSON " + url

    timeMap = subprocess.check_output(cmd, encoding='UTF-8')
    #timeMaps.append(timeMap)

    i+=1
    f = open('timemaps9and10.txt', 'a')
    if timeMap != '':
        f.write(timeMap + ",\n\n")
    print(i,': ',timeMap)

    #print(timeMaps)

#with open('testingTimeMaps.txt', 'w') as outfile:
    #json.dump(timeMaps, outfile, indent=)
```

Figure 3: jsonGetTimeMaps.py

```python
import json
import sys
from pprint import PrettyPrinter
from datetime import datetime
import matplotlib.pyplot as plt

pp = PrettyPrinter(indent=2, stream=sys.stderr)
with open('allTimeMaps.txt') as f:
  timeMaps = json.load(f)
  #pp.pprint(timeMaps)

memento_bins = [[] for _ in range(0,101)]

ageVsMementoSet = set()
i=0
for memento_block in timeMaps:
    uri = memento_block["original_uri"]
    listOfMementos = memento_block["mementos"]["list"]
    numberOfMementos = len(listOfMementos)
    firstMementoDate = memento_block["mementos"]["first"]["datetime"]
    today = datetime.today()
    birthday = datetime.strptime(firstMementoDate, "%Y-%m-%dT%H:%M:%SZ")
    estimatedAge = today - birthday
    daysOld = int(estimatedAge.days)
    #if numberofMementos > 0:
    if numberOfMementos < 100:
      ageVsMementoSet.add((daysOld, numberOfMementos))

    i+=1
    #print(f"{i:} -> {numberOfMementos:} -> {estimatedAge}")

    idx = numberOfMementos // 100
    memento_bins[idx].append((numberOfMementos, uri))

x,y = zip(*ageVsMementoSet)
plt.scatter(x, y, s=5)
plt.show()
bin_size = 100

for idx, sub_list in enumerate(memento_bins):
    bin_start = idx * bin_size
    bin_stop = idx * bin_size + (bin_size - 1)

    #print(f"{idx:} -> ({bin_start} to {bin_stop}) {len(sub_list)}")

```

Figure 4: analyzeTimeMaps.py

list of mementos, and the number of mementos (length of the list of mementos). I used the python datetime import to estimate the age of each URI (line23).

To create Figures 5, 6, and 7, I created a list of bins sized 100 from 0 to 10,099(line 12 and line 37). For each URI, based on the number of mementos, I placed a tuple into it's corresponding bin which included the original URI and the number of mementos (line 32). To determine where to place the tuple, I divide the number of mementos by bin size. The remainder will have the index of the bin where the tuple should be placed. At the end of this process, I printed the results to the console (Figure 5) so that I could construct my graphs by hand (Figures 6 and 7). The results were that 706 of the URIs had no mementos, 294 had at least one, and 263 of the 294 had between 1 and 99 mementos (Figure 7). To address the homework question directly, this collection of URIs is not well archived, seeing as over two-thirds of the collection had no mementos, and about one-fourth of the collection only had between 1 and 99 mementos. The difference between the 'At least 1' and the '1-99' bars from Figure 7 (31 mementos) is what you see in Figure 6's bar chart. These links could be considered outliers, since they only make up 31 of the 1000 links. (Please note that the last bar of the chart should have the label 10,000-10,099. It got cut off.)

```
0 -> (0 to 99) 263
1 -> (100 to 199) 7
2 -> (200 to 299) 3
3 -> (300 to 399) 1
4 -> (400 to 499) 1
5 -> (500 to 599) 1
6 -> (600 to 699) 3
7 -> (700 to 799) 1
8 -> (800 to 899) 1
9 -> (900 to 999) 0
10 -> (1000 to 1099) 1
11 -> (1100 to 1199) 1
12 -> (1200 to 1299) 0
13 -> (1300 to 1399) 1
14 -> (1400 to 1499) 0
15 -> (1500 to 1599) 0
16 -> (1600 to 1699) 0
17 -> (1700 to 1799) 1
18 -> (1800 to 1899) 0
19 -> (1900 to 1999) 0
20 -> (2000 to 2099) 0
21 -> (2100 to 2199) 0
22 -> (2200 to 2299) 0
23 -> (2300 to 2399) 1
24 -> (2400 to 2499) 0
25 -> (2500 to 2599) 0
26 -> (2600 to 2699) 0
27 -> (2700 to 2799) 0
28 -> (2800 to 2899) 0
29 -> (2900 to 2999) 0
30 -> (3000 to 3099) 0
31 -> (3100 to 3199) 0
32 -> (3200 to 3299) 0
33 -> (3300 to 3399) 0
34 -> (3400 to 3499) 0
35 -> (3500 to 3599) 0
36 -> (3600 to 3699) 0
37 -> (3700 to 3799) 0
38 -> (3800 to 3899) 0
39 -> (3900 to 3999) 0
40 -> (4000 to 4099) 0
41 -> (4100 to 4199) 0
42 -> (4200 to 4299) 0
43 -> (4300 to 4399) 0
44 -> (4400 to 4499) 0
45 -> (4500 to 4599) 0
46 -> (4600 to 4699) 0
47 -> (4700 to 4799) 0
48 -> (4800 to 4899) 0
49 -> (4900 to 4999) 0
50 -> (5000 to 5099) 0
51 -> (5100 to 5199) 0
52 -> (5200 to 5299) 0
53 -> (5300 to 5399) 0
54 -> (5400 to 5499) 0
55 -> (5500 to 5599) 0
56 -> (5600 to 5699) 0
57 -> (5700 to 5799) 0
58 -> (5800 to 5899) 0
59 -> (5900 to 5999) 0
60 -> (6000 to 6099) 0
61 -> (6100 to 6199) 0
62 -> (6200 to 6299) 0
63 -> (6300 to 6399) 0
64 -> (6400 to 6499) 0
65 -> (6500 to 6599) 0
66 -> (6600 to 6699) 0
67 -> (6700 to 6799) 0
68 -> (6800 to 6899) 0
69 -> (6900 to 6999) 0
70 -> (7000 to 7099) 0
71 -> (7100 to 7199) 0
72 -> (7200 to 7299) 0
73 -> (7300 to 7399) 0
74 -> (7400 to 7499) 0
75 -> (7500 to 7599) 0
76 -> (7600 to 7699) 0
77 -> (7700 to 7799) 0
78 -> (7800 to 7899) 0
79 -> (7900 to 7999) 0
80 -> (8000 to 8099) 0
81 -> (8100 to 8199) 0
82 -> (8200 to 8299) 0
83 -> (8300 to 8399) 0
84 -> (8400 to 8499) 0
85 -> (8500 to 8599) 0
86 -> (8600 to 8699) 0
87 -> (8700 to 8799) 0
88 -> (8800 to 8899) 0
89 -> (8900 to 8999) 0
90 -> (9000 to 9099) 0
91 -> (9100 to 9199) 0
92 -> (9200 to 9299) 0
93 -> (9300 to 9399) 0
94 -> (9400 to 9499) 0
95 -> (9500 to 9599) 1
96 -> (9600 to 9699) 0
97 -> (9700 to 9799) 0
98 -> (9800 to 9899) 0
99 -> (9900 to 9999) 0
100 -> (10000 to 10099) 7
```

Figure 5: Bins 0-5,699

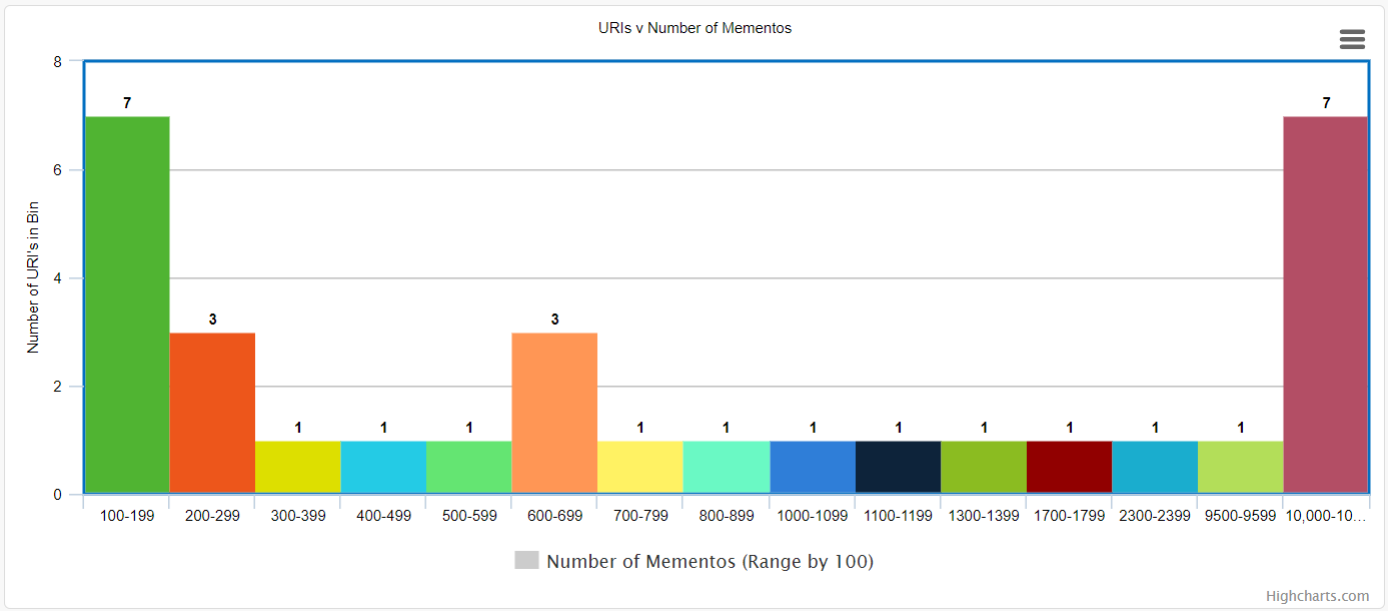

Figure 6: Number of URIs x Number of Mementos (Outliers)

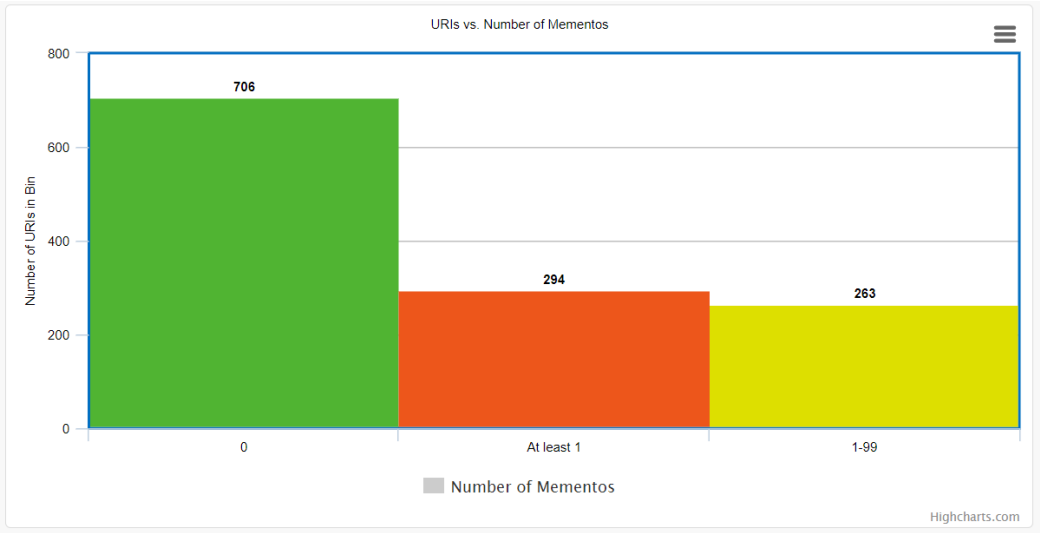

Figure 7: Number of URIs x Number of Mementos

To answer Q3, I used python's matplotlib library. In the analyzeTimeMaps script (Figure 4) on line 25-26, I added a tuple to a set to include the age of the URI in days and the number of mementos. Between lines 34-37, I created the data for the scatter plot, created the plot, and then displayed it. I created two separate scatter plots to better show the data. Figure 8 contains every URI that had a memento, but we cannot see a relationship due to the outliers with thousands of mementos or very old age. To create Figure 9, I used the conditional on line 26 to exclude mementos with more than 100 mementos. This seemed appropriate based on the earlier observation that of the 294 URIs with at least one memento, 263 of those were less than 100. Looking at the scatter plot of Figure 9 though, there is not much to be said. You would think that the older a page, the more mementos it would have (positive correlation). But, looking at Figure 9, there is really no discernible line to show that relationship.

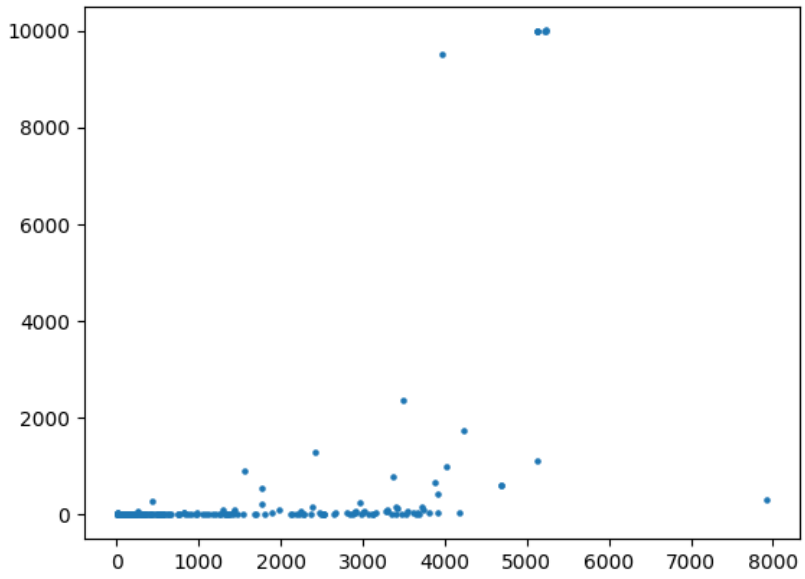

Figure 8: Effect of Age on Number of URIs (for URIs with at least 1 memento)

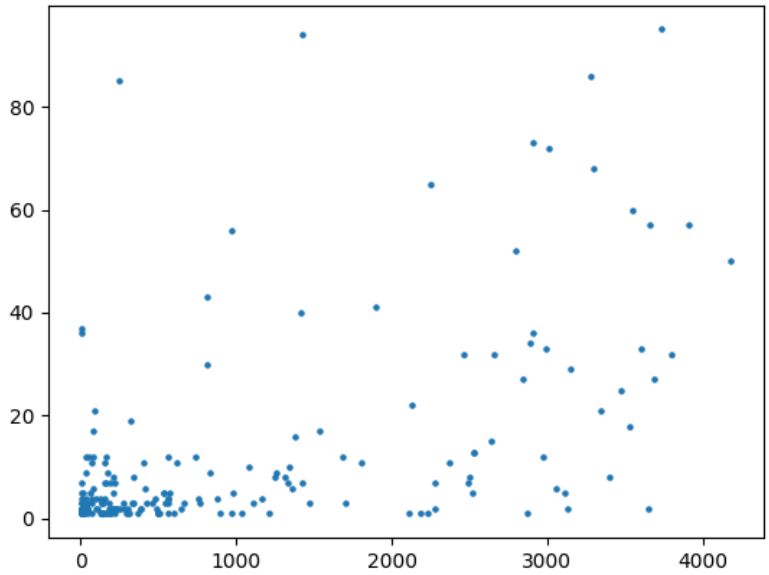

Figure 9: Effect of Age on Number of URIs (for URIs with less than 100 mementos)

Question 4

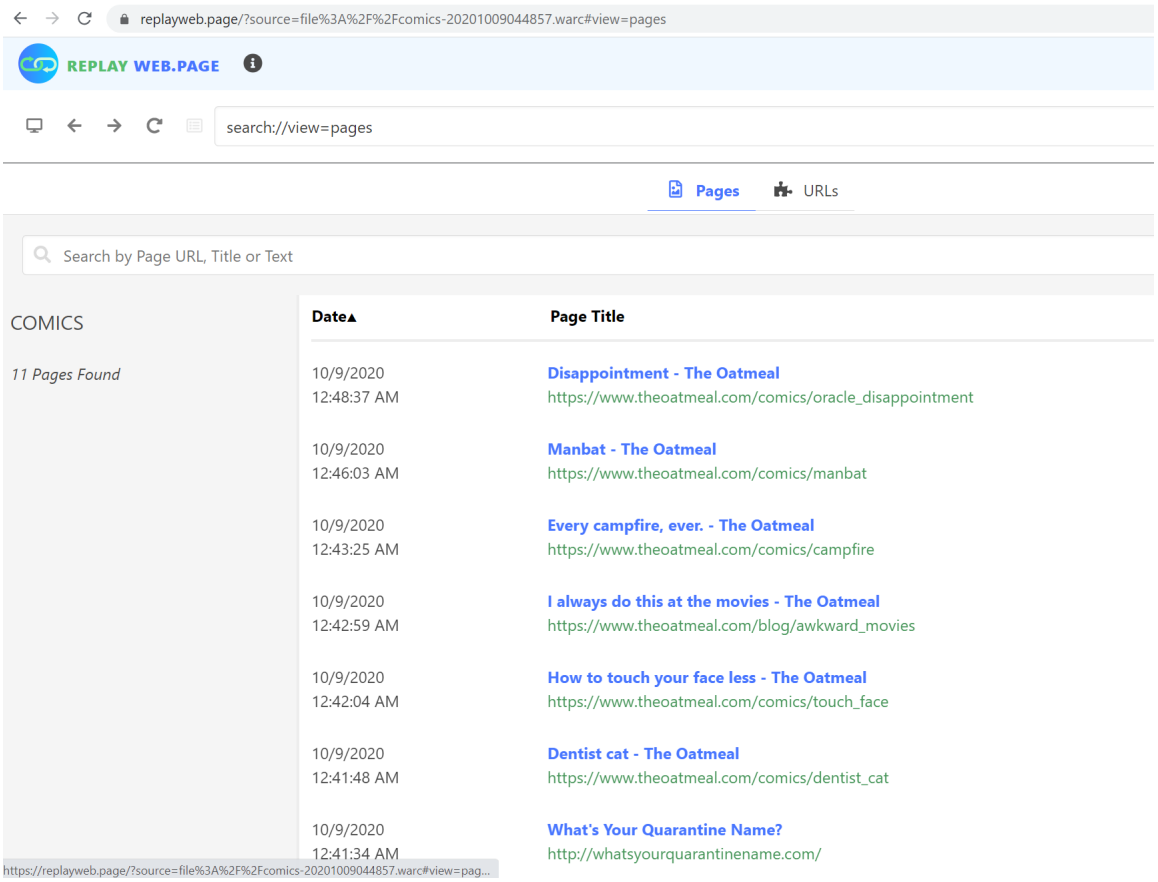

Figure 10: Proof of WARC

Question 4

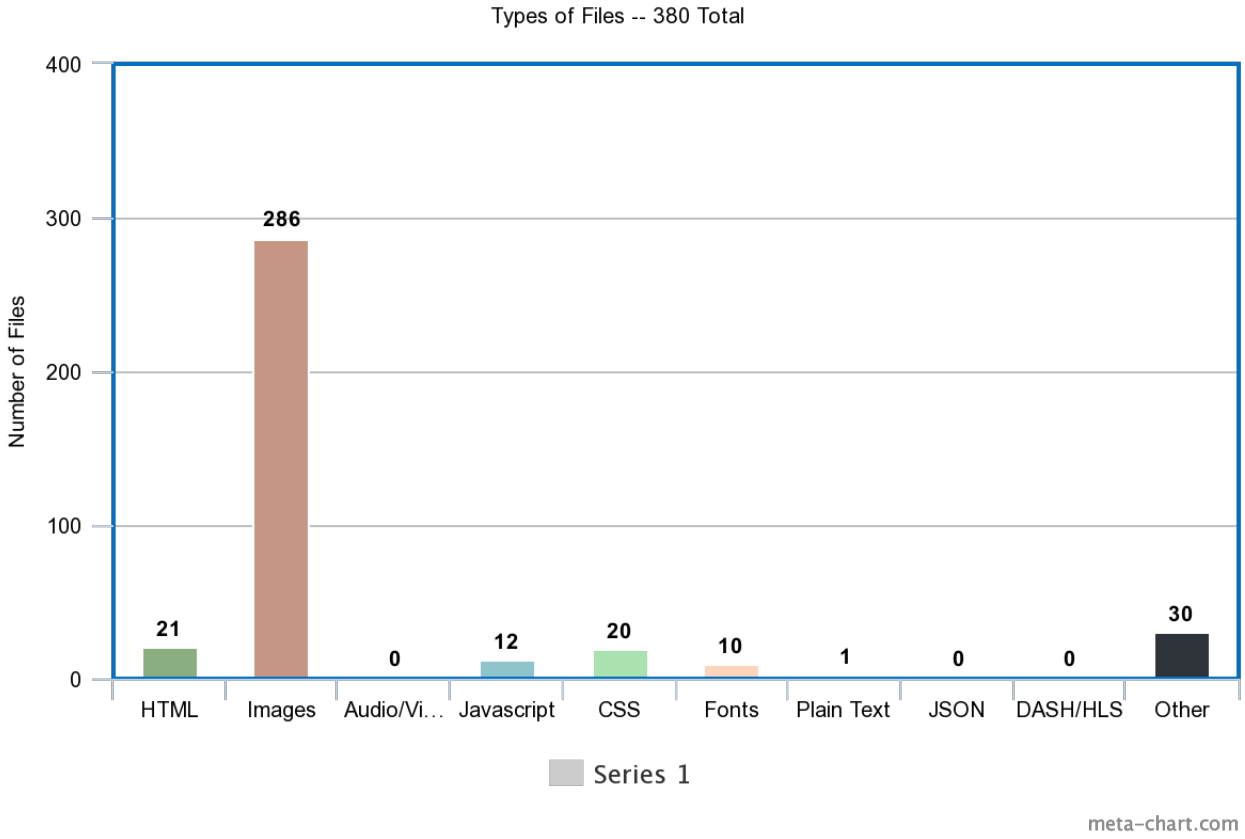

Figure 11: Types of Files

I decided to archive web pages from a website called The Oatmeal. I archived comics from the author of the page. The archives look just like the original page. I had first tried to archive The Onion pages, but they were large and the archives did not look great, so I chose the above topic instead. Another issue I had was learning to work with Conifer's UI. It seemed straight forward, but I had to manually stop each capture when I thought it was complete. I am still not sure if I was using the service as intended, but I got my archive collection!

Figure 10 shows that I was able to download the WARC file from Conifer and upload it to relay.web.page for analysis. Figure 11 shows the distribution of file types. Most of the files were images, which did not surprise me, as I archived comics.

References

1. 6.10.2 timedelta Objects. (2006, October 18). Retrieved October 11, 2020, from https://docs.python.org/2.4/lib/datetime-timedelta.html

2. Kaushik, N. (n.d.). Converting Strings to datetime in Python. Retrieved October 11, 2020, from https://stackabuse.com/converting-strings-to-datetime-in-python/

3. Saullo G. P. Castro. (2018, April 27). Make scatter plot from set of points in tuples. Retrieved from https://stackoverflow.com/questions/17478779/make-scatter-plot-from-set-of-points-in-tuples