OLD DOMINION UNIVERSITY

Assignment 2

David Bayard

February 18, 2019

QUESTION 1.

Write a Python program that extracts 1000 unique (collect more e.g., 1300 just in case) links from Twitter. Omit links from the Twitter domain (twitter.com). Also note that you need to verify that the final target URI (i.e., the one that responds with a 200) is unique.

Solution:

The Twitter API offers users the ability to extract tweets from Twitter, by automating GET requests and responding with Tweet objects. Using this API resulted in JSON formatted Tweets, which could then be scanned for URIs.

Although, target links were only available for the past 7 days using this API, thus popular queries were used including “Sports”, “Trump”, “Republicans”, and “Immigration”. The snippet of code listed below displays the Search method from the Tweepy API.

```
for tweet in tweepy.Cursor(api.search, q="Immigration -filter:retweets",count =
    tweetsCountMax, include_entities=True).items(maxNumTweets):
```

Listing 1: Tweepy API Search Method

Once the Tweets have been gathered, the URI’s are extracted by accessing the “expanded url” attribute of all the Tweet objects. The collected URI’s are filtered for uniqueness by deduplicating and extending the URI’s. Issues were encountered in this portion of the assignment, where shortened links were not completely extended. The code below implements a loop which will call “filterLinks-Func(link)”, which will follow redirections and return a URI and status code. A counter was implement to prevent an infinite loop from occurring.

```
    try:
      link, statusCode = filterLinksFunc(URL)

      if(statusCode == 200):
        newList.append(link)

      else:
        counter = 0;
        while(statusCode != 200 and counter != 5):
          filterLinksFunc(link)
          counter = counter + 1

    except:
      genericErrorInfo()
```

Listing 2: Extending URI’s

After extending all of the URI's, a deduplication function was applied to remove duplicate functions and guarantee uniqueness. This was accomplished by breaking each URI into its seperate components and comparing it to other existing URI's. Using this same concept, the domain names of "Twitter.com" were removed.

```
    for URI in unsanitizedList:
      scheme, netloc, path, params, query, fragment = urlparse(URI)
      toCheck = netloc + path


      if toCheck not in deduplicatedList:
          deduplicatedList.append(URI)
  except:
    genericErrorInfo()

  return set(deduplicatedList)
```

Listing 3: Deduplicating URI's

QUESTION 2

Download the TimeMaps for each of the target URIs. We'll use the ODU Memento Aggregator

Solution:

In order to download the TimeMaps for each target URI, Docker was installed and the MemGator image was downloaded. This allowed requests to be made via the local machine, rather than making requests through the ODU server. As stated above, the TimeMaps were downloaded by making GET requests via the localhost, which were sent to port 1208 of the server. The specific command used: sudo docker continer run –rm -it -p 5000:1208 ibnesayeed/memgator server

```
##### Make Request and store the JSON response in json_obj

      requestToMake = "http://localhost:5000/timemap/json/" + URI

      resp = requests.get(requestToMake, timeout=30)

      ##Write response in json format ##
      json_obj = resp.json()
```

Listing 4: Make TimeMap Request

Once a JSON object is created from the request, it is possible to read the response, and parse the data as shown.

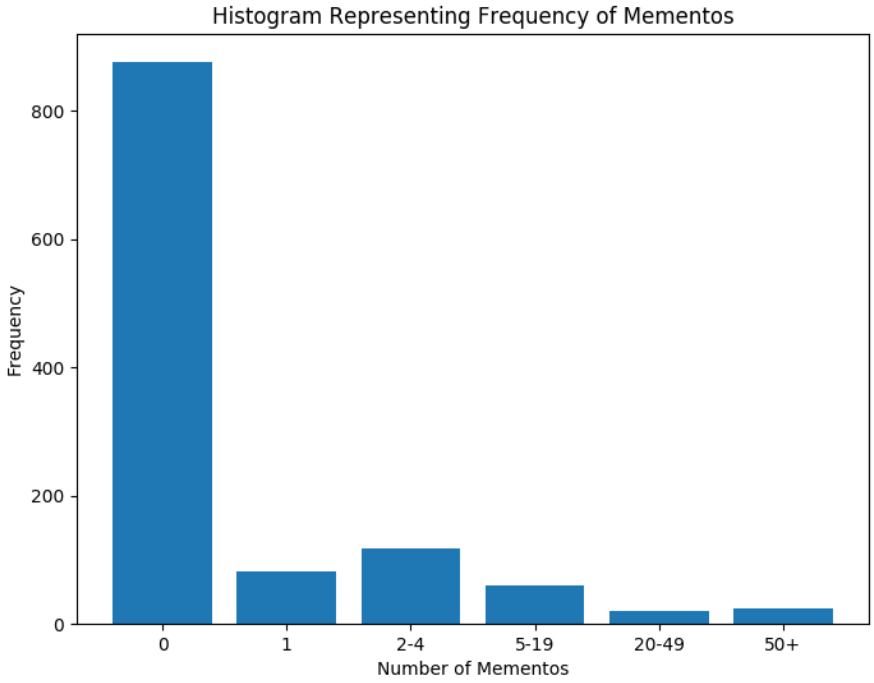
```
##### Make Request and store the JSON response in json_obj
      dataGood = {"URI" : "value", "Mementos" : "value2", "TimeMapFileName" : "value3" }

      dataGood["URI"] = URI
        dataGood["Mementos"] = len(json_obj["mementos"]["list"])
        dataGood["TimeMapFileName"] = fileName
```

Listing 5: JSON Request

The code above displays that the JSON format provides access to the content of the response, which is parsed earlier using the "resp.json()" function. This methodology is used through the entire program, making it simple to store and retrieve data from files, simplifying the program. Moreover, after extracting the JSON data, it is convenient to append the data to a dictionary, and create a list of dictionaries. Then, this list of dictionaries can be written to a .json file, and can easily be accessed later in the program.

```
allMementosText.write("URI: " + dataGood["URI"])
        allMementosText.write("\n")
        allMementosText.write("TimeMapFileName: " + dataGood["TimeMapFileName"])
        allMementosText.write("\n")
        allMementosText.write("Mementos: " + str(dataGood["Mementos"]))
        allMementosText.write("\n\n\n")

        #allMementos is file being written to
        json.dump(dataGood, allMementos, indent=2)

        list.append(len(json_obj["mementos"]["list"]))
        print("Added " + str(len(json_obj["mementos"]["list"])))
```

Listing 6: Writing JSON to file



(a) Histogram representing URI vs Number of Mementos

QUESTION 3

Estimate the age of each of the 1000 URIs using the "Carbon Date" tool:

The exact same methodology from question 2 can be applied here to get the estimated age of each URI. This is achieved by launching the Docker container for the “Carbon Date tool” using the command “sudo docker container run –rm -it -p 8888:8888 oduwsd1/carbonate -s”.
With the Carbonate container running, each request is processed through Carbonate, and every response is stored within the “CarbonRecords.json” file.

```
1
2  for URI in listURI:
3
4    try:
5
6      requestToMake = "http://localhost:8888/cd/" + URI
7      resp = requests.get(requestToMake)
8
9
10
11   except:
12     genericErrorInfo()
13
14   try:
15     json_obj = resp.json()
16
17   except:
18     genericErrorInfo()
```

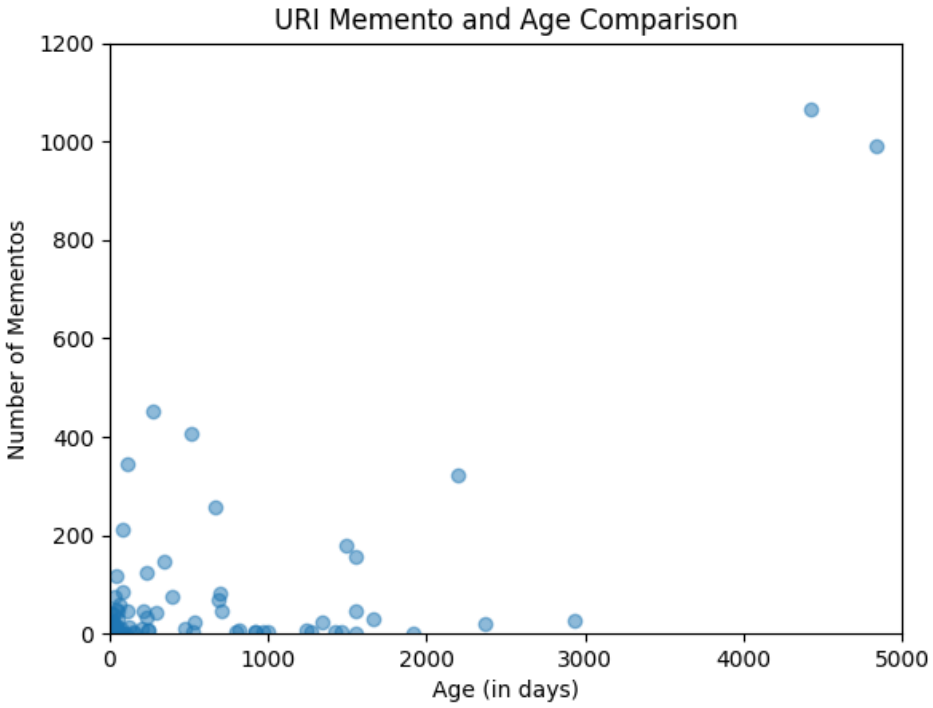
Listing 7: Make Carbon Date Request

Using the “CarbonRecords.json” and “MementosTrack.json” files, a scatter plot is created representing the number of Mementos, and the estimated creation date of the URI corresponding to the URI.

```
    for link in URIBoth:
  for a in carbontoData:
    if(a["URI"] == link and a["Carbon-Date"] != ""):

      toParse = a["Carbon-Date"]
      carbonDate = dateutil.parser.parse(toParse).date()

      age =  abs((today - carbonDate).days)


      ageVsMemento["Age"] = age


      for b in mementoData:
        if(b["URI"] == link):
          ageVsMemento["MementoCount"] = int(b["Mementos"])
          ageVsMementoList.append(ageVsMemento.copy())
```

Listing 8: Number of Mementos vs Age in days

The main function of the code above is to parse the date from each URI that has a Carbon-Date, as well as the number of Mementos corresponding to the URI. This method is not efficient, but the design of the program prevented a more efficient approach. In the future, a preferred course would be to write all of the data to a single JSON file, making the process of parsing simple. Regardless, the data was gathered and the results were used to create the scatter plot depicted below.



(a) Scatter Plot of Memento Count vs Age of URI

In total, the results for the URIs, creation dates, and number of Mementos is categorized as shown below:

Total URIs: 1179
Number of URIs with Mementos: 303
Number of URIs with Estimated Creation Dates:1017

This information states that out of 1179 URIs, 1017 of them had estimated creation dates, and only 303 of them has at least one Memento.