Overcoming Social Media API Restrictions: Building an Effective Web Scraper

Nicholas B. Harrell[1], **Iain Cruickshank**[2], **Alexander Master**[2]
[1]Purdue University
Center for Education and Research in Information Assurance and Security
West Lafayette, IN 47907 USA
nharrel@purdue.edu
[2] Army Cyber Institute
West Point, NY 10996 USA
iain.cruickshank@westpoint.edu, alexander.master@westpoint.edu

Abstract

As social media platform application programming interfaces (APIs) are becoming more restrictive and costly to use, there is a considerable risk that researchers will be unable to address research problems related to online discourse. This paper presents a detailed examination of an effective approach to scraping X (formerly Twitter) data, leveraging the Selenium WebDriver for automated interaction with web pages. This technique circumvents the limitations of X's dynamic content generation and JavaScript-dependent interface, providing a robust alternative to traditional API-based data retrieval methods. By emulating human navigation patterns, this method offers insights into extracting real-time social media data, including tweets, likes, and retweets, which are crucial for various analytical applications.

Introduction

Recent trends of social media websites restricting API access, along with the growth of inherently segmented social media sites (e.g., Telegram) present new challenges for data collection for research. Previous API restrictions frequently sought to reduce the potential harm that large-scale data analytics could pose, such as the Cambridge Analytica scandal (Tromble 2021; Axel Bruns and Bruns 2019; Wiley 2019). However, when these API restrictions were put in place, they often had exceptions for academic research. More recently, an increasing number of social media sites have sought to restrict access even to academic researchers and to have only paid APIs available. An example is what happened to Twitter's API when it became X (Durbin 2023). Perriam et al. describe our current epoch as the post-API era, a characterization that resonates with those accustomed to the once unrestricted access to social media platform APIs (Perriam, Birkbak, and Freeman 2020). This delineation is evident when examining the scarcity of literature on specific types of social media time-series studies in recent years. Until approximately 2019, researchers could analyze numerous social media platforms, typically requiring nothing more than a free user account (Perriam, Birkbak, and Freeman 2020). To overcome these difficulties, we propose using a headless browser-based approach, which simulates a human user, to acquire data from social media sites. Specifically, we demonstrate a suite of techniques utilizing Selenium WebDriver, a tool designed for automating web browser interaction, to collect social media data for research.

Problem Space: Difficulties with Web Scraping

Typically, a web page can be easily scraped by something like the requests package.[1] Additionally, modifying the request header can also overcome issues with blocking automated web page access, which is frequent with many social media sites. However, scraping data from social media platforms presents additional unique challenges due to their dynamic nature, often relying on JavaScript for content generation. Unlike traditional websites, social media platforms frequently update content based on user actions or new data, dynamically altering their HTML structure. This dynamic content loading (DCL), facilitated by JavaScript, poses a significant obstacle to scraping efforts, as simple HTTP requests do not execute JavaScript and may fail to retrieve desired information. Consequently, researchers must resort to more sophisticated methods, such as leveraging parsing libraries (e.g., BeautifulSoup) in combination with HTTP requests, to navigate these challenges. However, these approaches require continuous script updates to adapt to changes in the site's structure or content, disrupting data collection efforts. Additionally, accessing dynamically loaded content (typically triggered by user interaction like scrolling) is particularly challenging from a terminal perspective, necessitating advanced skills in web design for effective navigation. Despite these limitations, diligent and skilled researchers can still extract valuable information through these means, albeit with increased effort and potentially reduced data quality as demonstrated by (Landers et al. 2016; Hillen 2019).

Methodology

To begin the discussion of our proposed methodology, we note that social media platforms need to allow traffic for humans to function; thus, if we approach data collection autonomously by simulating human behavior, we can continue to achieve our goal of data collection. Additionally, previous work has leveraged tools like BeautifulSoup (an advanced

[1] Requests packages or libraries are often built into terminals to allow users to interact with the internet with no graphical display.

HTML and XML Parser)[2] and Selenium[3] in tandem (Mancosu and Vegetti 2020a; Chapagain 2019; Mehta and Pandi 2019). The current state of media platforms calls for a more systematic method for scraping. Media platforms often enforce traffic policies to prevent denial-of-service (DOS) or fraudulent activity. However, with scraping for research, we are primarily concerned with violating X's rate limit and losing our ability to search. Selenium offers the capability to interact with dynamic objects on websites and we acknowledge we are not the first to use Selenium for scraping; however, we have not seen any viable approaches that handle the nuances of using scraping on dynamic media platforms. This section will highlight how to build a scraper that takes into account the features that could potentially trigger a platform's traffic policies, resulting in denial of request (or potentially being banned) from a platform.

Our methodology for scraping X data using Selenium WebDriver involves a series of steps designed to automate a web browser, mimicking human interaction to access and extract web content dynamically rendered by JavaScript. This section delves into the technical aspects of the scraping process, including interaction with the Document Object Model (DOM), CSS selectors, and handling JavaScript-loaded content.

Inspection Tools These are typical items you will find within the Inspect feature of Firefox or Chrome web browsers:

- **Developer Tools (DevTools) in Browsers**: Use the Elements panel to inspect the HTML structure and the Console to test JS functions.
- **Network Tab**: To observe requests made by JS functions, especially `fetch()` and `XMLHttpRequest()`, to load data dynamically.
- **Sources/Debugger**: To step through JavaScript code execution and understand how and when certain elements are manipulated or data are loaded.

Concept of Scraping

Web scraping, in general, follows a particular flow for data retrieval. However, in our design, we illustrate where the decision to collect and delay are in Figure 1. The specific research question drives the elements that need to be collected. Collecting more than necessary creates larger data compilations and requires more storage. We recommend conducting a full investigation for useful objects on sites as one will find that information can be replicated in multiple elements of the code. We found that most of the objects on X are stored within a single data structure. Depending on the research design, it may be important to consider the "time of the post" and the "time of the collection" if you are doing some kind of time-series analysis.

As shown in our Figure 1, we recommend storing a cookie to maintain long-term access. With the cookie loaded each time in the webdriver, it is easy to navigate directly to your target site based on the search query. Search query structures are unique to the platform and X is transparent about how their advanced search queries operate.[4] The easiest way to design a search query is to go to the advanced search page as in the footnote. Then, structure your query accordingly; afterward, copy the URL and plug in your variables accordingly in the script. X has a limit on the number of days that can be searched in a single request; therefore, you will need to build a script that adjusts the dates you want to search. We decided to do daily scrapes; then, move forward by one day. However, this was for a specific research problem. Some research may require daily pulls to avoid losing potentially removed ("taken down") content.

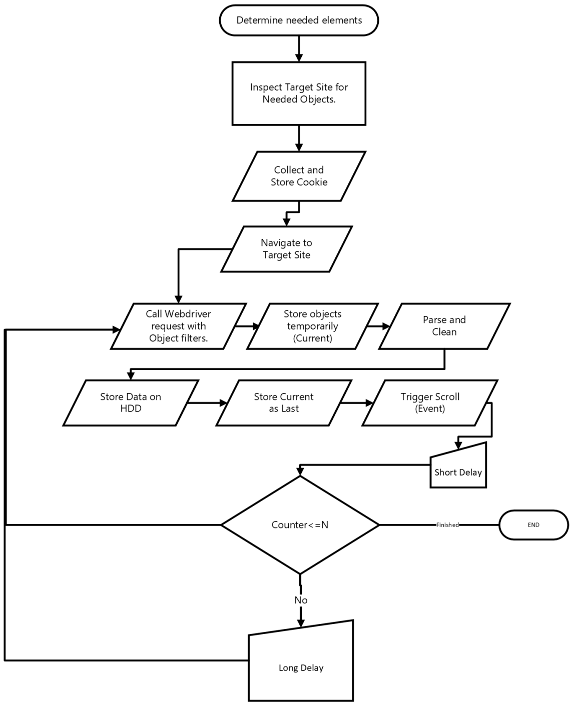


Figure 1: Concept of Scraping

In the scraper script, it is important to have variables for storing the previous epoch for comparison to prevent duplicates. We found that it was easy to pull one post and then, compare it to the last entry to ensure they were different. You can use any set of features to do this comparison.

We recommend parsing and cleaning while pulling data to prevent having to spend extra time in the future. For some problems, it may be easier to gather, then parse. It depends on the research. We provide examples in Appendix B. Here, we stress that the process of collection could take days, weeks, or months, depending on the research need. After parsing, you will dump your objects into a long-term storage object (e.g., json, xml, yaml, or csv file). Many research projects use storage objects, such as the json files used by the Open Observatory of Network Interference (OONI) to track censorship events, and used in research projects such as (Master and Garman 2023). Then, set your temporary

[2] https://code.launchpad.net/beautifulsoup/

[3] https://www.selenium.dev/

[4] https://twitter.com/search-advanced?lang=en

variable as last. Finally, trigger your scroll event.

We recommend adding a short delay at this point. This delay will prevent you from making requests too quickly and allow your program to finish smoothly. At this point, you will be cycling your process; N in Figure 1 is how many times you will cycle before a long delay is triggered. This long delay is determined by understanding the platform's threshold for denying a request. The purpose of the long delay is to prevent the platform from tracking your query request. One has to consider the user volume that a platform such as X or Meta experiences from moment to moment. It would be computationally infeasible for these platforms to maintain a queue of every user's request. These platforms use algorithms to track anomalous behavior. Therefore, if you provide enough delay, you can circumvent these heuristic-based blocking procedures.

Lastly, realize when you trigger an event such as a scroll or a click, the response does not have to be the same. Sometimes, you might pull an advertisement, popup, or an overlay feature. We will discuss how to circumvent these obstacles in the next section.

This section provides a general overview of how we recommend the scraper to flow. We discuss some variations and considerations in the next section. There are several ways to run the script; we offer our flow as a contribution to the community as it has worked successfully for our research ends.

Design Considerations

To web scrape effectively while avoiding detection, it is important to mimic human behavior. Our approach follows the philosophy that computers should imitate human speed. Understanding a website's threshold for detecting automation is essential. Social media platforms, designed for human interaction, often monitor interaction rates to prevent bots. Researchers with ample time can scrape slowly to evade detection, but those on tight schedules may face challenges. However, for faster data retrieval, one needs to test delays in website access to find the threshold where platform restrictions are avoided. In our research, a one-second delay with a two-minute pause after every sixty pulls proved effective.[5] These delays ensure that the scraper will not violate X's rate limit policy. X's policies do not forbid automated collection using third-party applications. Some conditions are disallowed; however, research was not included. The main heuristic measure that X monitors is rate. If you violate the rate limit, your searchability will be temporarily halted.

Collecting data daily and iteratively is essential to avoid missing valuable pieces relevant to one's research goals. Platforms often have limitations on how far back data are displayed (i.e., 7-day window), and posts can frequently be taken down or retracted. Along with continual collection, continually saving data (or dumping it) during the scrape is also a good practice.

Error handling is important. We use "try-except" blocks in our script to bypass errors. Try-except is a method of trying to run a specific block of code; instead of the program crashing with an error output, it handles the error by bypassing it. One should log the errors because sometimes we find that due to where the browser was scrolled, our scraper tried to capture an advertisement. Ideally, you want the scraper to skip over any potential disruptions. In our experience, we have not found retries to be successful. One can skip the scroll on the errors and attempt to recollect. However, we have found that the majority of the time, we either receive an error due to triggering the platform's traffic policy or we are attempting to collect an object that does not exist where the browser window is currently placed.

System interruption signals can happen due to memory constraints or some type of disruption. These interruptions are most likely caused by having arrays or lists in your script that are storing too much data in memory. Therefore, we recommend iterative dumping as shown in Figure 1. Ensure that your script knows how to handle signals and captures the data collected in long-term storage before exiting. This notion is sometimes referred to as "exiting gracefully."

Discussion

We have shown how to build a scraping tool for X, which can be adapted to other platforms. We discussed the challenges with parsing and several design considerations. These practices are approachable and develop with time; however, the further important discussion is an ethical one. This type of parsing will quickly prove to be useful and one will quickly realize this can be applied in many different settings. Just because we can, does not always mean we should.

Cambridge Analytica vividly demonstrated the potential misuse of publicly available information. As with any ethical research endeavor, scholars bear the responsibility to safeguard the well-being of those they study (Mancosu and Vegetti 2020b; Fiesler and Proferes 2018). The question of whether collecting social media data warrants Institutional Review Board (IRB) often oversight sparks debate. Ethical and legal concerns, as highlighted by (Mancosu and Vegetti 2020b), prompt reflection on the distinction between public and private data and the potential privacy implications of research endeavors. Instances such as (Zimmer 2020) underscore the risks associated with using publicly available Facebook data, which may compromise personal information and consumer privacy. Additionally, when collecting data on contentious or controversial topics, researchers must consider the potential ramifications, including unintended harm to individuals.

In scholarly discourse, the ethical presentation and dissemination of research findings, especially those aggregating information for thematic analysis akin to advertising companies' interests, are paramount. Ensuring non-traceability requires a controlled approach to sample representation and stringent restrictions on data sharing with researchers who uphold equivalent ethical standards. The recurrent removal of outdated datasets underscores the importance of ethical considerations in data management across disciplines. Furthermore, the discourse extends to data storage practices, revealing a notable disparity in concern across various disciplines, including law, psychology, and biology, as opposed to online research domains.

[5] Note that effective delays will vary by platform, and may change over time on a given platform.

The challenge lies in balancing the public accessibility of datasets with ethical considerations regarding representation and utilization, regardless of the data format. Ethical boundaries become pertinent when discussing the algorithms employed for data analysis. Advocating for the adoption of random unique identifiers as surrogates for usernames aims to preserve pseudonymity and prevent back-tracing to individuals. This methodology ensures that data analysis, particularly in natural language processing involving lexical pattern analysis, maintains individual privacy while facilitating the extraction of research-relevant information. Emphasis on reproducibility mandates that ethically managed datasets enable subsequent researchers to validate findings without compromising data integrity or participant anonymity. Thus, the primary concerns revolve around ethically gathering data to safeguard individual privacy and applying research methodologies that prevent direct association of results with the original dataset.

We bring the ethical discussion into this article to ensure that readers understand the consequences of poor data handling practices. We also want to show readers that when ethical considerations arise, we have an obligation to question the ethics of our practice. Lastly, we cannot claim ignorance when it causes detriment to another individual. Regardless of legality, researchers have an ethical obligation to do their due diligence.

Conclusion

In this paper, we demonstrate how to automate the collection of research data using Selenium and contemporary web scraping best practices. We show how this can be an effective approach and an alternative to using restricted APIs from website owners. We also discuss ethical concerns and potential consequences surrounding data collection through web scraping.

Acknowledgements

We would like to thank the Army Cyber Institute (ACI), Purdue Military Research Initiative (PMRI), and the Purdue Center for Education and Research in Information Assurance and Security (CERIAS) for their continued support in our research endeavors.

References

Axel Bruns; and Bruns, A. 2019. After the 'APIcalypse': social media platforms and their fight against critical scholarly research. *Information, Communication & Society*, 22(11): 1544–1566. MAG ID: 2956617434.

Chapagain, A. 2019. *Hands-On Web Scraping with Python: Perform advanced scraping operations using various Python libraries and tools such as Selenium, Regex, and others*. Packt Publishing Ltd.

Durbin, B. 2023. X changes its API to retire legacy tiers and endpoints.

Fiesler, C.; and Proferes, N. 2018. "Participant" perceptions of Twitter research ethics. *Social Media+ Society*, 4(1): 2056305118763366.

Hillen, J. 2019. Web scraping for food price research. *British Food Journal*, 121(12): 3350–3361.

Huggins, J. 2004. About selenium. https://www.selenium.dev/about/. Accessed on April 6, 2024.

Landers, R. N.; Brusso, R. C.; Cavanaugh, K. J.; and Collmus, A. B. 2016. A primer on theory-driven web scraping: Automatic extraction of big data from the Internet for use in psychological research. *Psychological methods*, 21(4): 475.

Mancosu, M.; and Vegetti, F. 2020a. What You Can Scrape and What Is Right to Scrape: A Proposal for a Tool to Collect Public Facebook Data. *Social Media + Society*, 6(3): 2056305120940703.

Mancosu, M.; and Vegetti, F. 2020b. What you can scrape and what is right to scrape: A proposal for a tool to collect public Facebook data. *Social Media+ Society*, 6(3): 2056305120940703.

Master, A.; and Garman, C. 2023. A Worldwide View of Nation-state Internet Censorship. In *Free and Open Communications on the Internet*, 1–21. Proceedings on Privacy Enhancing Technologies.

Mehta, S.; and Pandi, G. 2019. An Improving Approach for Fast Web Scrapping Using Machine Learning and Selenium Automation. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 8(10): 434–438.

Perriam, J.; Birkbak, A.; and Freeman, A. 2020. Digital methods in a post-API environment. *International Journal of Social Research Methodology*, 23(3): 277–290.

Tromble, R. 2021. Where have all the data gone? A critical reflection on academic digital research in the post-API age. *Social Media+ Society*, 7(1): 2056305121988929.

Wiley, C. 2019. *Mindf*ck: Cambridge Analytica and the plot to break America*. Random House.

Zimmer, M. 2020. "But the data is already public": on the ethics of research in Facebook. In *The ethics of information technologies*, 229–241. Routledge.

Appendix A: Website Related Key Terms

This section lays out web-related elements that are important for understanding the layout of a website.

HTML Elements The data we are looking for is usually embedded in a specific section of the <body> of the HTML; however, every media site will have a different web layout or cascading style sheets (CSS), which is a template for the basic layout of all their websites within a specific web domain. Learning the layout of the underlying CSS makes finding the resources on the website much easier.

- `<div>`: Often used to contain major content blocks or features on a page, such as tweets, user information, or navigation bars.
- `<span>`: Frequently used for text that needs to be styled differently from its surrounding text, like counts for likes and retweets.
- `<a>`: Anchor elements, which are used for links. This includes user profiles, hashtags, or links within a tweet.

- `<button>`: Used for interactive elements that users can click, such as the like, retweet, and reply buttons.
- `<input>` **and** `<form>`: For search bars or login fields. Useful if your script needs to log in or submit queries.
- `<ul>`**,** `<ol>`**,** `<li>`: Unordered, ordered lists, and list items, respectively. These might be used for comments, tweet threads, or navigation menus.
- `<img>`: For images. Profile pictures, embedded media in tweets.
- `<time>`: Elements specifically for timestamps, which can be crucial for dating tweets or other content.

JavaScript Functions These are the relevant functions that you will usually be searching for in javascript language usually annotated by (`<script> </script>`).

- `addEventListener()`: Essential for understanding how the page responds to user actions like clicks, scrolls, or key presses.
- `fetch()`/`XMLHttpRequest()`: Used to request data from a server dynamically. Observing these can help identify how new data (like tweets) is loaded.
- `setTimeout()`/`setInterval()`: Functions that delay actions or repeat them at intervals, possibly used for loading or updating content.
- `document.querySelector()` /`document.querySelectorAll()`: Methods to select elements from the DOM. Useful for pinpointing how scripts manipulate specific parts of the page.
- `window.scroll()`/`element.scrollIntoView()`: Functions to control scrolling, which could trigger the loading of additional content.
- `JSON.parse()`: Often used to parse JSON data received from server responses, which might include data for new tweets or updates.
- `history.pushState()`/`history.replaceState()`: Methods for manipulating the browser's history. Relevant for single-page applications (SPAs) that update the URL without reloading the page.

Appendix B: Navigating Selenium, Finding Objects, and Building the Scraper

Understanding Selenium

The journey of Selenium began in 2004 when Jason Huggins developed what is known as the Core mode, initially dubbed ”JavaScriptTestRunner” (Huggins 2004). Selenium supports multiple programming languages, including C#, Ruby, Java, Python, and JavaScript, enhancing its versatility across various applications. Its primary purpose is to automate web browser functions, a concept often referred to as ”headless browsing.” With knowledge of a website's schema and template, Selenium allows users to identify and interact with specific web elements relevant to their research, enabling automated script modifications for element extraction.

Users of this tool should understand that it operates by treating a web browser as a driver, where the script provides input commands that control the webdriver. The webdriver, in turn, can extract and collect outputs through dynamic web interactions. Instead of visualizing content for the user, the output is the retrieval of web objects and elements.

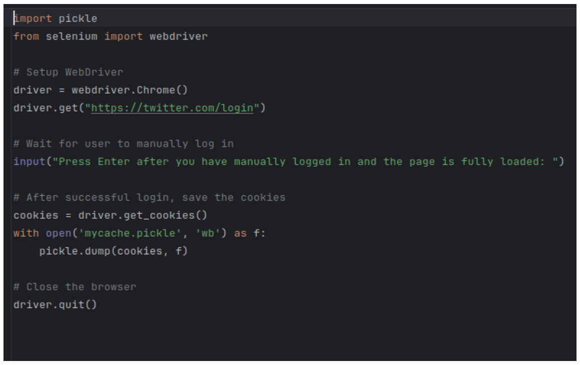
However, it is important to note that excessive use or rapid event triggering could surpass rate limits or violate platform policies, potentially resulting in the loss of interaction capabilities. The Selenium Project explicitly does not support CAPTCHA bypassing, as CAPTCHAs are designed to deter automation, making attempts to circumvent them unethical. The goal is to create tools that align with platform policies without generating excessive traffic.

Utilizing virtual private networks (VPNs) often leads to CAPTCHA challenges; thus, it is advisable to ensure that your VPN's exit node(s) are not flagged for abuse. Practices such as using a proxy server or VPN hopping are detectable through browser headers and account associations with the platform, indicating that behaviors resembling malicious activity, should be avoided. For more information on activities discouraged by the Selenium Project, please visit their website.[6]

Building your tool

We only use X as a use case for our demonstration. Every site will have a different scheme and template.

The first thing we have to consider is login and authentication. X does not specifically require it; however, one issue we ran into is having a valid cookie to bypass Twitter's homepage. The easiest way to bypass this issue is to build a cookie by logging in and capturing the cookie in a pickle file. Another alternative is developing a selenium profile, which allows your driver to maintain it's own cookies and past login history like an actual human interactive browser. This method is helpful if you are scraping multiple platforms.

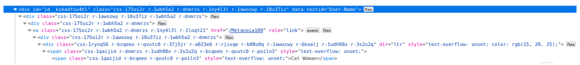


```
import pickle
from selenium import webdriver

# Setup WebDriver
driver = webdriver.Chrome()
driver.get("https://twitter.com/login")

# Wait for user to manually log in
input("Press Enter after you have manually logged in and the page is fully loaded: ")

# After successful login, save the cookies
cookies = driver.get_cookies()
with open('mycache.pickle', 'wb') as f:
    pickle.dump(cookies, f)

# Close the browser
driver.quit()
```

Figure 2: Capturing Login in Pickle

Referring to Figure 2, we can see a basic script to capture the pickle file. Using Selenium webdriver, we open up a browser that navigates to the twitter login. Then, once the page is fully loaded we press a button to get passed the input in the script. Then we call the get_cookie() function on the browser and dump it in pickle format. There is a way to do this with JSON as well.

[6] https://www.selenium.dev/documentation/test_practices/discouraged/



```
▼<div id="id__kskadfsu4hl" class="css-175oi2r r-1wbh5a2 r-dnmrzs r-1ny4l3l r-1awozwy r-18u37iz" data-testid="User-Name"> flex
 ▼<div class="css-175oi2r r-1awozwy r-18u37iz r-1wbh5a2 r-dnmrzs"> flex
  ▼<div class="css-175oi2r r-1wbh5a2 r-dnmrzs"> flex
   ▼<a class="css-175oi2r r-1wbh5a2 r-dnmrzs r-1ny4l3l r-1loqt21" href="/Metanoia100" role="link"> event flex
    ▼<div class="css-175oi2r r-1awozwy r-18u37iz r-1wbh5a2 r-dnmrzs"> flex
     ▼<div class="css-1rynq56 r-bcqeeo r-qvutc0 r-37j5jr r-a023e6 r-rjixqe r-b88u0q r-1awozwy r-6koalj r-1udh08x r-3s2u2q" dir="ltr" style="text-overflow: unset; color: rgb(15, 20, 25);"> flex
      ▼<span class="css-1qaijid r-dnmrzs r-1udh08x r-3s2u2q r-bcqeeo r-qvutc0 r-poiln3" style="text-overflow: unset;">
        <span class="css-1qaijid r-bcqeeo r-qvutc0 r-poiln3" style="text-overflow: unset;">Cat Woman</span>
```

Figure 3: Finding CSS Objects

From this demonstration, it is easy to see some of the flexibility that Selenium allows. Since this method is primarily focused on using Selenium rather than investigating themes on social media, we will demonstrate how we find a specific object (e.g., a CSS Object) and use it with Selenium. We do not list all examples of our specific implementation here; one must investigate their specific platform's query and search policies to understand how to manipulate them. Our assumption is that our readers already have some basic coding experience and can infer how to collect the data once it is readily available as output. We will be using the theme ”#cats” for demonstration purposes.

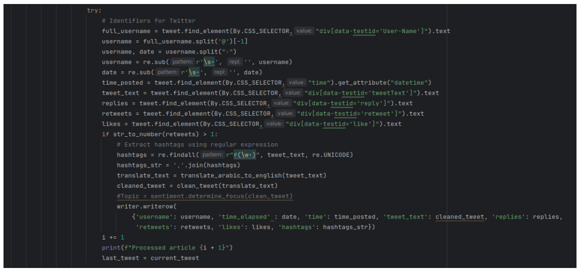
We must find where the objects are stored within the dynamically loaded content. With investigation, we see that many objects are stored in a TAG called 'article.' This is where we will have to manipulate our script to capture all of these objects, we can do this by using the web drivers find_elements called as such: **tweets = driver.find_elements(By.TAG_NAME, 'article')**. Since elements are plural, this creates a list of all the articles found within the current browser view. This idea alone is problematic because this means as we scroll, we could potentially capture the same articles twice, causing duplication. Therefore, we must consider a way to prevent duplication of copying articles. One way we can alleviate this is by putting a condition in your tweets loop that checks if the current tweet equals the last tweet. Another approach is to check for duplicates in your tweets list; however, this list will get overwritten each time you make a new request; therefore, we use the former method. An alternative would be to store your lists and compare the lists for duplicates. There are advantages and disadvantages to each approach that balance time and space.

Now that we can capture articles, we must figure out what objects exist within the articles. If we refer to Figure 3, we can see that the objects for a post are stored in a CSS Object ”div[data-testid='User-Name'].” User-Name is the first hint to where the other objects might be stored. With further investigation, we found a test-id object for tweetText, reply, like, retweet. The time object was an object by itself.

We now have all the basic elements for creating our own script. We demonstrated how to store a cookie, inspect a website for elements, use the elements in a Selenium webdriver, and write the objects for research use. With these basic fundamentals, anyone with basic scripting experience will be able to find the information they are looking for if the platform allows it. Figure 4 shows how one would use these objects in a script for capture.

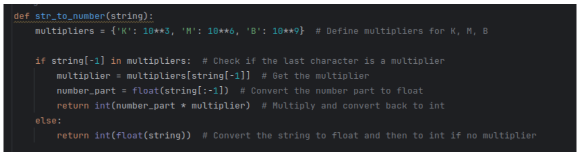
Parsing Objects

We briefly discussed parsing above; we elaborate on the issue in this section. Some problems will arise when you finally collect your data. Most researchers intend to do quantitative analysis on likes, retweets, and replies of social media datasets. However, you will find that after scraping and collecting your data, there will be strings that have K, M, or possibly B in the columns standing for thousand, million or billion, respectively. To streamline this, we want to design functions to handle this data.



```
try:
    # Identifiers for Twitter
    full_username = tweet.find_element(By.CSS_SELECTOR, value: "div[data-testid='User-Name']").text
    username = full_username.split('@')[-1]
    username, date = username.split("·")
    username = re.sub( pattern: r'\s+',  repl: '', username)
    date = re.sub( pattern: r'\s+',  repl: '', date)
    time_posted = tweet.find_element(By.CSS_SELECTOR, value: "time").get_attribute("datetime")
    tweet_text = tweet.find_element(By.CSS_SELECTOR, value: "div[data-testid='tweetText']").text
    replies = tweet.find_element(By.CSS_SELECTOR, value: "div[data-testid='reply']").text
    retweets = tweet.find_element(By.CSS_SELECTOR, value: "div[data-testid='retweet']").text
    likes = tweet.find_element(By.CSS_SELECTOR, value: "div[data-testid='like']").text
    if str_to_number(retweets) > 1:
        # Extract hashtags using regular expression
        hashtags = re.findall( pattern: r"#(\w+)", tweet_text, re.UNICODE)
        hashtags_str = ','.join(hashtags)
        translate_text = translate_arabic_to_english(tweet_text)
        cleaned_tweet = clean_tweet(translate_text)
        #Topic = sentiment.determine_focus(clean_tweet)
        writer.writerow(
            {'username': username, 'time_elapsed' : date, 'time': time_posted, 'tweet_text': cleaned_tweet, 'replies': replies,
             'retweets': retweets, 'likes': likes, 'hashtags': hashtags_str})
    i += 1
    print(f"Processed article {i + 1}")
    last_tweet = current_tweet
```

Figure 4: Capturing Data

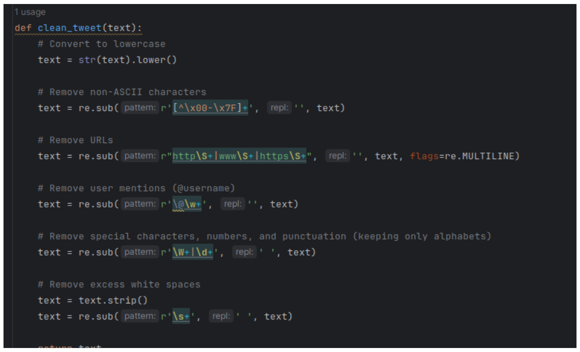


```
def str_to_number(string):
    multipliers = {'K': 10**3, 'M': 10**6, 'B': 10**9}  # Define multipliers for K, M, B

    if string[-1] in multipliers:  # Check if the last character is a multiplier
        multiplier = multipliers[string[-1]]  # Get the multiplier
        number_part = float(string[:-1])  # Convert the number part to float
        return int(number_part * multiplier)  # Multiply and convert back to int
    else:
        return int(float(string))  # Convert the string to float and then to int if no multiplier
```

Figure 5: Strings to Numbers

In Figure 5, we show a simple function for handling these types of strings. One can simply pass your likes, retweets, and replies through this function and get a numerical representation allowing for quantitative analysis.

Depending on what type of data you are scraping, one may run into different language characters. This problem is research-dependent. Whatever operating system we are using may provide constraints on how the text is displayed. We are able to overcome these problems by creating a function that does translation or creating a function that converts those characters to the appropriate characters. We recommend building single functions for each nuisance instead of trying to build one script that handles all. This method allows you to implement these functions as needed in different applications. One can just build these into a class object that you can call as you would a Python library.



```
1 usage
def clean_tweet(text):
    # Convert to lowercase
    text = str(text).lower()

    # Remove non-ASCII characters
    text = re.sub( pattern: r'[^\x00-\x7F]+',  repl: '', text)

    # Remove URLs
    text = re.sub( pattern: r"http\S+|www\S+|https\S+",  repl: '', text, flags=re.MULTILINE)

    # Remove user mentions (@username)
    text = re.sub( pattern: r'\@\w+',  repl: '', text)

    # Remove special characters, numbers, and punctuation (keeping only alphabets)
    text = re.sub( pattern: r'\W+|\d+',  repl: ' ', text)

    # Remove excess white spaces
    text = text.strip()
    text = re.sub( pattern: r'\s+',  repl: ' ', text)

    return text
```

Figure 6: Clean Tweet Function

One of the challenges with gathering data online with bot-infested forums is that they often have many URLs, hashtags (#), or random characters that disrupt our analysis. Realize that these are all just ASCII characters and they are all capable of being parsed out. The standard way to do this is through regular expression commands. Regex is important to learn for parsing data and narrowing in on what you want. As an example, we demonstrate our cleaning function in Figure 6. Notice that what we look for is very specific and is tailored to the specific type of data collection we are doing. Researchers new to their field or who have not closely studied comparable datasets will have trouble discerning which elements of their collection efforts are valuable. Researchers who study social media often need to make custom scripts for their research. Also, many of the tools found online in git repositories require updating or tweaking to ensure they function for your use case. Learning these techniques is important because many of your limitations in data collection may be caused by a lack of understanding. Learning basic coding in a scripting language (e.g., Python, Bash, Powershell, Perl, or Ruby) is essential for being able to build useful data representations.