Article

Social Science Computer Review
1-21
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/0894439319886019
journals.sagepub.com/home/ssc
SAGE

Trails of Data: Three Cases for Collecting Web Information for Social Science Research

Fumin Li[1], Yisu Zhou[1], and Tianji Cai[1]

Abstract
As the availability of online data grows rapidly, researchers are confronted with a pressing question: How should social scientists collect Internet data for research? This study focuses on one of the most commonly used data collection techniques: web scraping. Going beyond canned approaches by leveraging a general framework of data communication, this study illustrates how online information can be systematically queried and fetched for reproducible research. To generalize our approaches, we additionally explore the variations in site security and architecture that analysts may encounter during the scraping process before they are given access to the desired data. The approaches we introduce do not rely on any proprietary software and can be easily implemented on any computing platform with programming languages such as Python or R. The methodological discussion in this study is meant to be applicable to current web-based research efforts. We include three examples with complete Python implementation. We also present an integrated workflow that enables researchers to produce analytical data sets that are traceable and thus verifiable for analysis or replication. Lastly, options related to the validity and efficiency of data are discussed, and we highlight the ongoing debate surrounding the ethics of online data collection, ultimately advocating for the fair use of online data.

Keywords
data collection, reproducible research, web scraping, headless browser, APIs, Python

The field of quantitative social science has undergone an extensive transformation over the past decade (Salganik, 2017). The proliferation of quantifiable information available to researchers has led to the emergence of multiple subfields within the social sciences, one of which is computational sociology (Bainbridge, 2007), in which researchers harness a massive scale of data and computing power to reconfigure classic sociological studies. Accompanying the proliferation of data sources is the increased accessibility of computer power coupled with declining costs. However, applying computational approaches within the field of social science has not been without its critiques

[1] University of Macau, China

Corresponding Author:
Tianji Cai, University of Macau, Avenida da Universidade, Taipa, Macau, China.
Email: tjcai@um.edu.mo

(e.g., Crawford, Miltner, & Gray, 2014; McFarland, Lewis, & Goldberg, 2016). While discussions on the applicability, ethics, and affordances have been growing over the last 5 years (for a review, see Lazer & Radford, 2017), we feel that a major component of data collection within this vibrant field has been missing and as such aim to bring much needed attention to an increasingly popular data source: the Internet. To be sure, there is no shortage of recipes to harvest Internet data from a general (e.g., Kouzis-Loukas, 2016; Mitchell, 2018; Munzert, Rubba, Meissner, & Nyhuis, 2015) and social science perspective (Ignatow & Mihalcea, 2017), but in this article, we illustrate and discuss scraping data from the open Internet through a methodological lens.

We believe that the web scraping approach is noteworthy for two reasons. First, web scraping requires more of an engineering effort while at the same time insisting that the researcher possesses a deep understanding of the quantification process, thus arguably making such digital data collection quite a different undertaking than the more traditional off-line methods. With the fluidity of the online world, the ambiguous ethical boundaries of accessing data pose new challenges for analysts. Naturally, such an undertaking requires more education and open discussion than the field of social sciences currently offers. Currently, traditional methodological training courses typically focus on retrospective data collection from surveys and in-depth interviews.[1] And, while the development of more user-friendly computational software platforms has certainly benefited researchers in many respects, the "plug-and-play" technology has also impeded students and colleagues from developing the necessary in-depth knowledge of quantification to carry out more complicated tasks (Moran, 2005). No doubt, there are enthusiasts writing comprehensive documents on this very topic;[2] however, we are not interested in simply writing one more recipe to add to the plug-and-play catalog, but instead, aiming to demystify and make more rigorous the practice using a common framework.

The second motivation concerns debates among empirical quantitative social scientists regarding reputable research. As research design becomes increasingly complicated, it requires more effort from the research community to scrutinize, verify, and eventually reproduce the results. From an empirical point of view, we note that even though the replication of scientific findings is an integral part of the scientific research enterprise (Peng, 2009, 2011), the social science community has yet to accept reproducibility as the norm (King, 1995).[3] In recent years, social scientists from multiple disciplines have launched a debate about reproducibility (Anderson et al., 2016; Camerer et al., 2016; Christensen & Miguel, 2016; Gilbert, King, Pettigrew, & Wilson, 2016). The questions surrounding reproducibility puts a spotlight on the legitimacy issues that contribute to the lack of faith in social science research. Today, only a handful of journals enforce a strict data availability policy (Herndon & O'Reilly, 2016). As stated previously, in the digital world, the level of complexity in data collection arguably exceeds that in the analog world (Salganik, 2017). Hence, we are interested in making web scraping more transparent to the community by showing how data are/could be produced and reproduced.

We believe there is no simple answer to definitively address all the issues associated with online data collection. In this article, we focus on one increasingly useful approach for data collection: data scraping from the Internet. Our intention is to present to the social science community multiple existing web scraping techniques, and we argue that the existing methods can all be put under a unified framework for web data collection. We illustrate two principal scraping approaches and give specific steps for three examples: a collection of media releases from a metropolitan police department in the United States, posts on the social media site Twitter, and data gathered from a centralized database in China that contains court sentencing documents. These examples gradually increase in data collection complexity. Although bits and pieces of canned approaches to data scraping has existed for quite some time, our article is instructive in the following respects. First, it does not rely on any proprietary software and thus can be achieved on almost any computing platform with open-source software. Second, we are completely transparent with our collection strategy for all three cases, which makes the final data product traceable, verifiable, and reproducible. We hope to provide more transparency on how data are collected as well as clarity on the steps required to transform ready-made data into analytical data for research purposes. Third, the framework shown is generalizable to various scenarios of online data. Finally, for social scientists, it is important to note that acquiring the desired data is only a part of the design process, and acting responsibly in an area where the ethical boundaries are less clear is an equally important component of the overall research design. Keeping responsible data scraping in mind, we conclude with some practical advice from the existing literature on ethical best practices.

Background

Despite the benefits of having access to online data, the pure abundance of information has paradoxically acted as a limiting factor for researchers. To utilize Internet data for social science research, we found it easier to backwards-engineer the research query, imagining situations starting at the end of the data trail and working backward toward the initial creation of data production. That is, our research design typically starts from websites or individual web pages as they are displayed to the end users, and then we scale-up our queries to include all other possible data points with similar characteristics.

The Anatomy of Web Communication

Recall that the goal of web page data collection is to automatically browse and store the web page content and organize the information into analyzable data. But not all web pages are created equally. A web page can be classified as either static or dynamic depending on how information is stored and displayed (Frye, Plusch, & Lieberman, 2003). A static web page shows the same information for everyone, whereas a dynamic web page loads information based on a user's inquiry or behavior.[4] In other words, the content of a dynamic web page is generated dynamically within the web browser after the user interacts with the site, as opposed to a static web page where the web browser presents the same information, regardless of who is accessing the site. For instance, police departments in major metropolitan areas in the United States, such as Los Angeles, regularly release crime-related information to the general public along with other useful information, providing a great source for understanding crime and policing that can be easily located through the police department's newsroom section. Because each static web page is self-contained, analysts can manually save the web page and later extract the data within it. Comparatively, in China, courts are mandated to release detailed sentencing information to the official website, China Judgement Online (CJO), where the documents are uploaded and searchable to the general public. Yet, due to the scope of information for each case, CJO does not display everything from the documents on the returned query page. Once the user submits their search terms, the returned page proceeds to interact with specific browsing behaviors, such as scrolling down the page, and by showing a varying number of search results. As such, the construction and underlying architecture of the CJO website is different from their LA counterpart. The dynamic structure of the CJO pages does not allow a user to simply save each web page as an HTML file to pull the desired information.[5]

Although both of these examples are online sources, the ways that such data can be useful to analysts are very different. Once we ascertain what type of information we need (e.g., Landers, Brusso, Cavanaugh, & Collmus, 2016), we typically apply one of two approaches to collecting the data. In Approach 1, we send data fetch requests to the website's server and directly capture the response from the website. In situations like these, a server's responses are usually in a machine-readable data format and are stored in HTML, XML, or JavaScript Object Notation (JSON) files.[6] This approach is suitable for static websites with no authentication mechanisms. The appeal of such

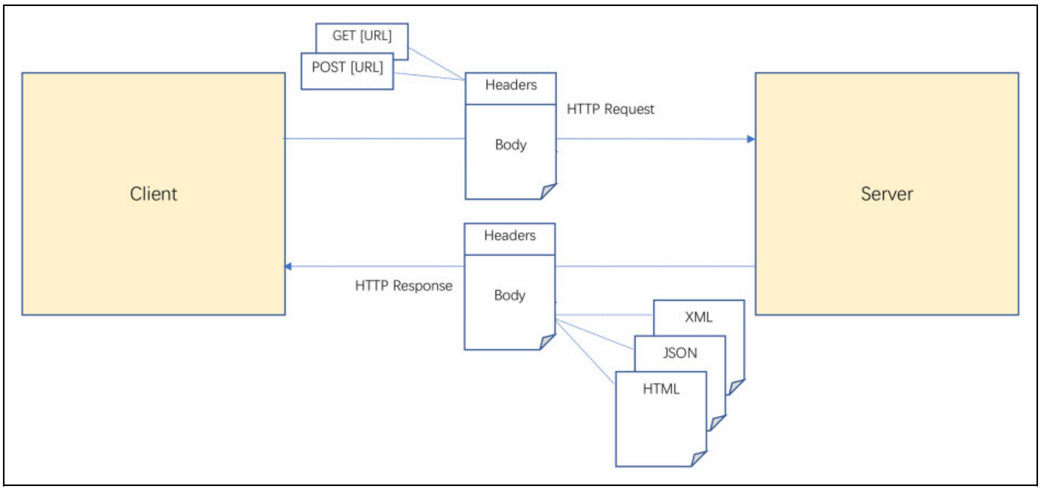

Figure 1. Client–server communication cycle.

an approach is in its simplicity and efficiency. It requires low technical know-how, and many canned programs can achieve this goal easily. But the limitation is obvious: It is very sensitive to the dynamic elements of the site as well as the website's anti-scraping mechanisms, a problem that is discussed further in the examples (see the next section for details).

To address this issue, we implement Approach 2, which is suitable for when the website is dynamic and/or has an authentication process. This strategy mimics how a human would browse a website and is robust against most anti-scraping mechanisms. The disadvantage is that it is relatively slow and less efficient. In addition, modern web pages contain many hidden components to track a user's location, behavior, and advertising preferences, which, while useful to the website owner, are not likely to be utilized by analysts. When capturing a complete web page, all the contents from a single page are fetched during data collection, making postprocessing burdensome. Therefore, analysts should determine beforehand which approach to use based on some initial inspection.

Instead of committing to any singular approach, it is important to understand how information is communicated over the Internet. In Figure 1, we illustrate how these two approaches could be understood under a single unified framework, namely, the client–server communication framework. On the left-hand side is the *client* computer that requests the data by pushing a *request* (e.g., a user clicking a blue link on a web page or a line of query command sent from a terminal), to the right-hand side *server* computer. The server receives the request and later sends the *response* to the client if the request is accepted by the server. When the requested information is ready, the information will be sent to the client via the response in machine-readable formats such as XML, JSON, and HTML. Under this client–server communication framework, Approach 1 is implemented before the web browser starts to render the data into a human-readable format by directly siphoning off the response data stream. Approach 2 works on the very end of the client side after all the responses are processed by the web browser into rendered HTML pages. Thus, Approach 1 typically results in nested, machine-readable data. A simple contrast of these two approaches is that Approach 2 works on the final product (i.e., the web pages) and Approach 1 works on the intermediary product (i.e., the response data).

The reason that this framework works is that as long as the information displayed on the website is intended for browsing, the transmission of information on the Internet complies with specific rules that both the client and the server follow.[7] For data collectors, we need to understand how to produce these requests in an automated fashion. In most cases, the common protocol used is http. Under this protocol, each request or response is uniformly organized, and the contents can be located and visited via a global address called the URL. Thus, the first step is typically for the client to establish a communication channel with the server to request certain content. Once such request is granted, the server will point the client toward a specific URL. With the URL, the web browser can then retrieve the content using http and store it in a structural format. When we send a request or receive a response for each URL, both will have a *header* component and a *body* component. The body of the response usually contains the data that we want to transfer through clients and servers. The header describes the request by listing how the data should be encoded, the type of formats, and who is requesting the data.

In static websites, if we want to retrieve all the news that the Los Angeles Police Department (LAPD) posted on their website, for instance, we can directly request the web pages that contain the news from its URLs and automate this request for other similar web pages quite easily. However, a simple URL request is not feasible if we want to download all homicide-related sentencing documents from the CJO website because unlike the LAPD news web page, the desired URLs for those CJO documents are hidden from us. For dynamic websites such as the CJO, the user originally has a frame web page with no specific content and needs to query the server using specific search conditions. Each query will then generate a request. If the request is accepted by the server, then a response will be delivered to the browser and the contents are injected into the frame page, rendered into human-readable format. However, the website will dynamically generate (or “refresh”) the page based on the user’s behavior such as scrolling down a page. Therefore, different users will see different web pages.

Anti-Scraping Protections

We demonstrated in previous sections that before the automation process can be implemented, the researcher first needs to locate the desired contents and understand how the contents are stored and delivered to the client. This is a crucial step to construct any trails of data due to the variety of web page architecture. Locating the contents is like following bread crumbs on a trail, and in that process, the researcher must circumvent a great deal of obstacles and distractions that many websites now deploy to guard against unwanted intruders; such mechanisms are collectively called anti-scraping protections.

Anti-scraping protections guard against nonhuman requests of data fetching initiated within a short period of time to avoid a server crash due to an overwhelming number of requests. In our case, a large number of data queries for research purposes would fit into this category, which many websites define as unwanted. Thus, analysts must understand how their data treasures are hidden and act accordingly.

The simplest website access control is to require proper authentication from the user before the website responds to any requests.[8] Once the analyst obtains proper credentials, their requests are formerly recognized by the server. There are a variety of authentication methods beyond simply username and password, such as using cookies or captcha (see the glossary). There is no standard recipe for authentications, and the researcher has to monitor the http communication to understand how the server allows access to certain client requests.

Like authentication, analysts need to be aware of other protection mechanisms such as limiting Internet protocol (IP) addresses, detecting robot web browsers, and tracking user behavior. The most common protection is to limit or terminate access based on the number of requests sent from a single IP address during a certain period of time. If the number of requests exceeds the limit, either further requests will be terminated or a captcha will be deployed to determine whether the request is sent by a human or a machine. To work around such protections, the researcher could either use a proxy server to change IP addresses after a certain number of requests have been made or add delays between requests to make them within the website's limit. Because analysts are the cause of extreme request loads on the server's end, we suggest making the scraping frequency within a certain limit as a sign of politeness (see Discussion section for detail).

Another major type of protection is based on the detection of the robot web browser. In a http request, the header field tells the server what the client wants to do, for example, retrieving or subscribing information, and a subfield of the header field, namely `User-Agent`, provides some basic information about the client such as the type of computer and detailed version information of each of the web browsers installed on the computer. Some websites use the information provided by the `User-Agent` subfield to grant discretionary access to a specific computer. Thus, to configure the right `User-Agent` subfield by each request would solve the problem. However, in more complex cases, certain websites generate a unique signature by combining the information from the client web browser, authentication, and IP address and embed it in http cookies while connected. Therefore, additional engineering efforts, such as cookies construction and modification of the `User-Agent` subfield, are needed to communicate with these sites.[9]

The cookies have a trove of information embedded within them. In the cookies stored, information from the user, such as log in, browsing history, and even clicking on a particular part of the page, can be detected. Constructing cookies requires some experience with the target website and is usually done after the analyst understands the entire operational routine of the authentication mechanisms. Some cookies are very complicated, as the codes for generating these are mixed into unreadable codes for humans.

The last category of protection is to track user behavior, such as the movement of a mouse cursor, to see whether the request is sent by a human. Tracking user behavior is commonly implemented along with the other protections mentioned above. Hence, it is not easy to get around, and efforts to identify and block specific ways of encoding the tracking request are required.

Arguably, it requires relatively high levels of technical know-how to successfully implement the above steps. As such, we recommend utilizing a *headless web page browser* as a pragmatic alternative. The headless web page browser is a web browser without a graphical appearance and is often used for web development such as analyzing website performance or debugging web applications automatically. From a data collection perspective, the headless browser automates human browsing behavior, including keyboard input and mouse movement to facilitate web data collection, hence not considered by websites as a robot user. However, one disadvantage of using the headless browser is that it demands high CPU and RAM resources. While the headless browser does not execute the graphical part of a website, it still needs to load almost everything else on it. Therefore, the running time is longer than that of a single http request. Thus, using a headless browser is usually considered as a last resort when anti-scraping mechanisms cannot be bypassed or computational cost is not a major concern.

Increasingly, many website owners also design their sites to respond to certain permitted requests to facilitate data gathering by a third party. This is usually referred to as an application programming interface (API). An API is essentially a canned version of data collection routines, protocols, and tools made by the website owners to facilitate data collection. The operation of an API is similar to a math function: Through http communication, the client only needs to supply the input to the server with specific parameters, and the server conducts the computations and prepares the queried data, then sends it back to the client. Many modern dynamic web pages transform the frame and the contents of a web page separately, such that the web page can just reload a partial portion of a page instead of the whole page. APIs usually provide the pure contents of the web page in XML or JSON, thus the responses from APIs usually contain few redundancies. From a data collection perspective, finding the APIs might be a good way to extract our desired data out of the redundancies.

Illustrated Examples

In this section, we illustrate the previous discussion with three examples: (1) LAPD's website, which is a static web page, to extract data from news releases; (2) Twitter's API to retrieve tweets according to search conditions; and (3) China's CJO website that implements a dynamic structure and complex anti-scraping measures. These three cases were chosen specifically to address the variation in website architecture and data accessibility hurdles that analysts will likely encounter. The LAPD case is static, and we demonstrate web scraping techniques using Approach 1. The Twitter case is dynamic but with ready-made tools to make it easy to retrieve data. We also applied Approach 1 to this case. The CJO case is dynamic and requires analysts to program their own pipeline with an existing tool kit; we use Approach 2 for this case. In terms of the data collection effort, these three cases progress in their complexity and difficulty. All cases were realized using Python language.

LAPD Example

The LAPD website is completely open to the public and doesn't require any authentication for http communication. There are no anti-scraping methods deployed. We use Approach 1 to directly send http requests and extract data from the received responses before they are rendered by the web browsers. Suppose we are interested in all of the released news archives from 2017, and we can obtain the 2017 news archive list from the LAPD at http://www.lapdonline.org/2017_archives, with each news archive organized by month. Since this web page is our first layer list web page, we extract the URLs of each month's archive from here and then visit each URL to get the second layer list of web pages (monthly archives, e.g., http://www.lapdonline.org/december_2017). In the second layer list, we can find the exact links to our interested news topic and then extract a brief description of each single news story as well as the URL to the web page of its full article that is our document web page. To facilitate the data collection process, two Python libraries are utilized: *Requests* (Version 2.20.1; Reitz, 2014) to deal with the http requests and responses and *Beautiful Soup* (Version 4.6.3; Richardson, 2015) to parse the received HTML data and subsequently extract data from the received HTML code.

Since the HTML data contain a series of tags to indicate its elements, we can parse the documents by locating specific tags. For instance, in the list web page of the LAPD 2017 news archive, all the tags that contain the monthly archive URLs are under the tag `<div class="span9">`, and each monthly archive URL is under a `<p>` tag stored in the value of the "`href`" attribute that is nested inside the start-sign of the `<a>` tag. Each second-layer web page of the monthly archives contains a list of news topics, and each piece of the news has a date, title, a category code, and a dedicated URL for the detailed article. Similar to the first-layer web page, the URLs for each of the detailed news articles are enclosed in the value of the "`href`" attribute under a `<p>` tag nested in a main tag (`<div class="span9">`). After successfully parsing the data, for example, detailed news, we store the extracted information into a two-dimensional data chart that uses columns as the variables and rows as the observations, following the tidy data convention (Wickham, 2014). All codes and annotations are included in the online repository.

Retrieving Tweets Using Twitter's API

Using Twitter to conduct social media studies in both academia and the greater industry has been a thriving endeavor in recent years (e.g., Blank, 2017). Compared to other platforms, Twitter's infrastructure allows any user to follow other accounts, allowing researchers to harvest almost 100% of their data through Twitter's APIs. While many commercial and free tools have been developed for researchers who have no prior technical or programming skills to collect data, finding appropriate tools can be time-consuming and challenging due to various issues such as operation platform, version, and specialized features. Here, we offer an example that directly utilizes Twitter's API for data retrieval that can be easily extended to other scraping requests for researchers with basic programming skills. For simplicity, we omit the steps that each user would use to first register a Twitter developer account in order to obtain the API.

The official standard Twitter search API allows the analyst to search tweets posted in the past 7 days, and the user must be authenticated before using the official Twitter APIs. Here, we submit a query to find tweets that contain key word "Donald Trump." We will use two pieces of information, the "API key" and the "API secret key" for authentication while sending requests to the Twitter server. The authentication process of Twitter's API is complicated if the analyst wishes to program the entire pipeline from scratch, but given Twitter's popularity, the pipeline has already been automated, greatly simplifying the process. To demonstrate our case, we chose the *Twython* (Version 3.7.0; McGrath, 2019) package. The authentication for Twitter's API requires two steps: (1) send the two pieces of information to gain an "Access Token" and (2) send the "API key" and the retrieved "Access Token" to get authenticated. After authentication, we can query the Twitter server, using "Donald Trump" as the query key word, "en"(English) as the desired language of the tweets, and get 100 tweets per request.

```
# auth_step_1, Get ACCESS_TOKEN by sending API_KEY and API_SECRET
twitter = Twython(API_KEY, API_SECRET, oauth_version=2)
ACCESS_TOKEN = twitter.obtain_access_token()

# auth_step_2, Get authenticated by sending API_KEY and ACCESS_TOKEN
twitter = Twython(API_KEY, access_token=ACCESS_TOKEN)

# Get 100 English tweets with keyword "Donald Trump"
twitter.search(q='Donald Trump', count=100, locale='en')
```

Box 1. The request for retrieving tweets from official Twitter application programming interface.

Since the official API limits how much time one can go back in history, analysts can also utilize Twitter's own advanced search function, which involves a bit more work to access Twitter data that are older than the normal limit (e.g., 7 days). Suppose we want to find tweets that contain the same key word "Donald Trump" from January 30 to January 31, 2019. After filling the dialogue boxes for the desired key word and dates on the advanced search page at https://twitter.com/search-advanced and clicking the search button, we scroll down to the bottom of the search results page to see all the results. If the developer mode is activated on the web browser (e.g., Firefox developer mode), we can see that while the cursor is scrolling down the page, the browser is simultaneously sending new requests to load more results. Among the different requests, the XML http request is one of the most essential because it directly transfers data between the web browser and the server in JSON format without reloading the entire page. Two parameters are important in the request: "`q`" that contains the search key word (e.g., "Donald Trump" in our case) and "`position`" that indicates the position where the search results should load. Since each of the responses contain 20 tweets from the search results, the server needs to know where to reload the results. If no value is given, the response will start from the beginning of the search results. Once the XML http Request (XHR) is granted, the raw tweets can be extracted from the received responses using Python library *Beautiful Soup*.

The following code constructs the XHR that retrieves tweets using this method. The variable `header` builds the headers for the request, which declares our identity honestly to the server. The variable `search_params` supplies the search conditions such as the key word and the starting and ending dates. At Line #10, we assemble the request by adding the header and parameters together. It also takes the response from the server after calling the `requests.get` function in the Python library *Requests*. Lines #11 and #12 extract the position for the next request and the raw tweets from the JSON object archiving the current response.

```python
1.  def get_json(since, until, keyword, position):
2.      headers = {
3.          'User-Agent': "For academic purpose, please contact us: XXX@XXX.XXX.",
4.      }
5.      search_params = {
6.          'q': keyword + " since:" + since + " until:" + until,
7.          'max_position': position
8.      }
9.      url = "https://twitter.com/i/search/timeline"
10.     r = requests.get(url, headers = headers, params = search_params)
11.     new_position = r.json()['min_position']
12.     tweets_HTML = r.json()['items_html']
13.     return tweets_HTML, new_position
```

Box 2. Retrieving tweets from Twitter with in-web page application programming interface.

By submitting a sequence of XML requests, the tweets that match the search conditions can be retrieved. For instance, the code below downloads and prints all tweets that contain the key word "Donald Trump" from January 30 to 31, 2019. The function `get_json` implements the code presented in Box 3. At the beginning, the `position` parameter is set to be null. The function `get_json` is called until the search results reach the end. The command `get_tweets` simply utilizes functions in the Python library *Beautiful Soup* to clean up the retrieved tweets.[10]

```python
position = ""
while True:
    response = get_json(since="2019-01-30", until="2019-01-31", keyword='Donald Trump', position=position)
    if len(response[0].strip()) == 0:
        print(response)
        break
    position = response[1]
    tweets = get_tweets(response[0])
    for key,value in tweets.items():
        print(key, value)
```

Box 3. Downloading tweets with key word "Donald Trump" from January 30 to January 31, 2019.

China's CJO Example

China's CJO is a very complicated case because it requires proper authentication by issuing specific signatures to each browsing session and regularly updates its anti-scraping protections. There are two ways to bypass these mechanisms. The most efficient way is to program a header subfield (e.g., `User-Agent`) in the same way as any ordinary web page browser would in the header field of the http requests, which requires the researcher to have a deep understanding of how the authentication and anti-scraping methods work on this website. Web developers typically achieve this by examining the requests and their cookies that a website will send to the browser. This is the key step, but also the most complicated one, because it requires knowledge of web application development to understand what is stored in the cookie and how to use such information to construct a legit http authentication. Once the researcher has that, Approach 1 is still a valid strategy, but we do not recommend this approach because from a research replication point of view, this method of data collection does not allow the information to be traced or verified. This is because CJO changes it authentication process almost monthly, meaning that whatever http requests previously worked will likely cease working after some time.

Based on this assumption, we suggest a less efficient, but equally workable, solution: to use the headless browser for automating human browsing behaviors. Basically, we write a program to view the web page for us. There is no need to sniff out which request will bring the desired response. Because we will extract data from the rendered web pages (thus, less efficient), what you see in the web page is what you are going to get in the later steps. Of course, the cost of using the headless browser is high CPU and RAM usage. With enough patience, this strategy would avoid triggering any anti-scraping methods because as far as the server is concerned, the requests are being sent by an ordinary browser with mouse and keyboard input. As such, no matter how often the website updates its anti-scraping mechanisms, our program rarely needs to be updated, as long as it behaves like any human browser. The only factor to consider is the requesting interval. Usually, the website determines whether the requests are from a robot user if the intervals between requests are too small.

For simplicity, only the major steps of using a headless browser are presented here, all the codes and a detailed step-by-step guide, such as how to set up a terminal environment and install dependencies, can be found in the online repository. Our example involves fetching all the murder cases on the CJO with a sentence date range of September 3 to September 4, 2018. To download those documents, we first need to initialize the headless browser that is interactively controlled by Python's *Selenium* package (Muthukadan, 2018), which directly manipulates the browser's webdriver to communicate with the server by simulating human browsing behavior. A webdriver (chromedriver is the webdriver for Chrome; geckodriver is for Firefox) is literally the "driver" of a headless browser, which allows the user to locate the elements (tags in the HTML source, e.g., buttons, input boxes, text boxes) in a loaded web page, and based on that, control the headless browser with mouse- and keyboard-instructive codes. All the procedures are manipulated by programming codes, but the codes are all about mimicking mouse clicking, keyboard inputting, and of course, extracting desired information from the rendered web page HTML source.

Running a web browser in the headless mode means all the regular web surfing, such as searching and subscribing, needs to be done without a graphic interface according to the structure of the http communications. The following code initializes Firefox in the headless mode.

```
1.  # construct an option of headless to the browser
2.  opts = FirefoxOptions()
3.  opts.add_argument("--headless")
4.  
5.  # create a webdriver to control a Firefox browser.
6.  # point to the position of geckodriver which is the programmable controller for Firefox
7.  driver = webdriver.Firefox(executable_path="geckodriver.exe", options=opts)
8.  driver.set_page_load_timeout(10) # set timeout for 10 seconds
9.  driver.set_script_timeout(10)
```

Box 4. Initializing Firefox in the headless mode.

Line #7 initializes the Firefox webdriver in the *Selenium* package, a Python interface that controls the web browser by programming. Together with the webdriver interface, the path to the webdriver software (i.e., geckodriver for Firefox) must be assigned. Lines #8 and #9 define the amount of time (10 s) allowed for page loading and script executing before reporting an error, respectively.

Suppose we are interested in all available murder cases on the CJO website sentenced from September 3 to September 4, 2018. After filling the search conditions for *Type of Crime* = 刑事案件 ("*xingshi anjian*" or "Criminal cases"), *Case* = 故意杀人罪 ("*guyi sharen zui*" or "Murder"), *Start Date* = 09/03/2018, and *End Date* = 09/04/2018 in the dialogue boxes on the search page of the CJO, and by clicking the search button, an http communication is initiated. In an ordinary web browsing event by a human, a user will click the buttons and scroll options to select the search conditions, they will also input characters via a keyboard into the boxes if necessary. These mouse-clicking and keyboard-inputting behaviors can all be programmed into a headless browser. In order to tell the headless browser to execute an instruction, we must locate the element first. The webdriver allows many ways to locate an element, the most convenient being XML Path (XPath) language. XPath is essentially a path syntax that locates the tags in an XML document, and it can also be used in an HTML document. In developer mode of Chrome and Firefox, the XPath of an element can be generated by selecting the option in its mouse right-click menu. The selecting behaviors of search conditions can be constructed with the following code. The date conditions can be selected via mouse behaviors, but it is easier to input the start and end dates with a keyboard. When all search conditions are selected or input, we can then locate the search button and send a click instruction to it. Once the request is granted, the browser will redirect to a search result page with a list of sentencing documents that satisfy the search conditions. The corresponding URLs for the listed documents can be found by analyzing the HTML source code of the results page, just as in the LAPD example. In addition to XPath, the webdriver allows us to locate an element with its attribute value (e.g., id, class). To locate multiple documents in the search results, it is better to use `class_name` instead of XPath, since we should construct XPath manually for multiple elements.

```python
# Select Cause of Action as "故意杀人罪" (Murder)
CAUSE_XPATH = [
    '//*[@id="_view_1540966814000"]/div/div[1]/div[1]',
    '//*[@id="_view_1540966814000"]/div/div[3]/div[1]/div[2]/div/div[1]',
    '//*[@id="1"]/i',
    '//*[@id="161"]/i',
    '//*[@id="162_anchor"]'
]
for x in CAUSE_XPATH:
    driver.find_element(By.XPATH, x).click()
    time.sleep(0.5)

# Select Case Type as "刑事案件" (Criminal Case)
TYPE_XPATH = [
    '//*[@id="selectCon_other_ajlx"]',
    '//*[@id="gjjs_ajlx"]/li[3]'
]
for x in TYPE_XPATH:
    driver.find_element(By.XPATH, x).click()
    time.sleep(0.5)

# Input start and end date
DATE_START_XPATH = '//*[@id="cprqStart"]'
DATE_END_XPATH = '//*[@id="cprqEnd"]'
driver.find_element(By.XPATH, DATE_START_XPATH).send_keys('2018-09-03')
driver.find_element(By.XPATH, DATE_END_XPATH).send_keys('2018-09-04')

# Press the Search button
driver.find_element(By.XPATH, '//*[@id="searchBtn"]').click()
```

Box 5. Submitting a search request for murder cases sentenced from September 3 to September 4, 2018.

After we get the HTML of each document in the results list, we can extract some basic information and the URL link of each document. With the known URLs, a simple loop of http requests can retrieve each of the sentencing documents by pushing a GET request for each of the URLs.

Next, a loop of `GET` requests among the document links can be sent to the server that archives the sentencing documents to fetch the sentencing documents. To avoid terminations due to detected nonhuman requests, we also need to limit the number of requests by adding a random pause between consecutive requests, change the IP addresses by rotating among different proxies, and imitate human behavior by simulating mouse scroll and click. Finally, all extracted data will be stored tidily in the format that most data scientists use.

```python
# At search result webpage, set the option for showing 15 cases per page
select = Select(driver.find_element(By.XPATH, '//*[@id="_view_1545184311000"]/div[8]/div/select'))
select.select_by_visible_text('15')

# Scrape the search result
docList = list()
while True:
    # Get the list for current page
    docList_raw = driver.find_elements(By.CLASS_NAME, 'LM_list')
    # Scarping each case info
    for case in docList_raw:
        title = case.find_element(By.CLASS_NAME, 'caseName').text
        docType = case.find_element(By.CLASS_NAME, 'labelTwo').text
        link = case.find_element(By.CLASS_NAME, 'caseName').get_attribute('href')
        court = case.find_element(By.CLASS_NAME, 'slfyName').text
        docCode = case.find_element(By.CLASS_NAME, 'ah').text
        docDate = case.find_element(By.CLASS_NAME, 'cprq').text
        judgeReason = case.find_element(By.CLASS_NAME, 'list_reason').text
        docList.append({
            'title': title,
            'docType': docType,
            'link': link,
            'court': court,
            'docCode': docCode,
            'docDate': docDate,
            'judgeReason': judgeReason
        })
    # Get the "Next Page" button
    nextPage = driver.find_element(By.LINK_TEXT, '下一页')
    # Click the "Next Page" button if it was clickable
    if nextPage.get_attribute('class') != 'disabled pageButton':
        nextPage.click()
        time.sleep(2)
    else:
        break
```

Box 6. The complete process of retrieving sentencing documents from the China Judgement Online.

Discussion

In previous sections, we introduced a general framework for collecting open data from the Internet. We illustrated two approaches from this framework with three examples, and the final data set is in compliance with tidy data (Wickham, 2014), which most analysts would immediately recognize. We can save our acquired data into a plain text format that can be imported into any analytical workflow for downstream analysis. In this section, we discuss some of the challenges involved.

Validity, Efficiency, and Politeness

It is worth some discussion to address the validity of our code. In the rapidly changing world of the Internet, we are not only confronted with new websites or web services being added every day but with older websites updating themselves with the latest technologies. Along the way, new anti-scraping protections are deployed, arguably the biggest hurdle for researchers. Experienced analysts are constantly monitoring communication sessions to adapt their own pipelines. In addition, static sites such as the LAPD are becoming rare. Many websites that are currently built around user-generated content will begin to load dynamically according to different user behaviors. Even some static sites, such as the Chicago Police Department's newsroom section, increasingly use sophisticated scripts to generate report-like pages in the format of pdfs. Nonplain text content will add some complications for scrapers.

As the Internet continues to advance, complex modern websites will be more like the CJO. Just within the time period of our project, the website's anti-scraping mechanisms were upgraded at least 4 times in its major components and countless times in its smaller ones. With each upgrade, new obstacles were introduced for the analysts to tackle. Undeniably, hurdles like these show the fluid nature of collecting web data. Now, changes in website security are rolling out not in months, but in weeks, or even days. In this sense, web scraping requires more of an engineering effort than social science has traditionally demanded. As a result, in our experience, users should not view our provided codes as a foolproof option for data collection but a starting point for learning to build a pipeline by oneself. The example codes may be outdated soon, but the logic behind the steps to analyze the structure of a website will remain true for a much longer period of time.

A related concern is about the efficiency of data collection. For smaller sites such as the LAPD, a modern computer could fetch all the abovementioned data in a matter of minutes, depending on network connection speed. But for larger sites, such as the CJO or Twitter, the volume of data is huge. To increase efficiency, web data collectors regularly use IP proxies, which are services that can offer hundreds of thousands of unique IP address. The researcher can then deploy their scraping programs through each individual IP address to communicate with the data server to coordinate a data collection. But this option will multiply the client computer's computational load (particularly in the case of the headless browser, which is an option we recommend in the dynamic setting) as well as the server's traffic. Users should anticipate that beforehand.

Ethics

The last aspect of this type of research concerns the ethics of web scraping. Ironically, this should typically be the first consideration, but it is usually either buried deep in the recipe (e.g., Mitchell, 2018) or completely missing from it (e.g., Bonzanini, 2016). Currently, the ethics surrounding web scraping are ambiguous, at best. For engineers, the question of "should we do it?" always seems to come after "how can we do it?" as witnessed by a burgeoning publication market filled with web scraping recipes, of which we have listed quite a few in this text. On the other side of this debate is the emerging literature that largely falls under the name of "critical data studies" (Iliadis & Russo, 2016). This camp points out that conducting research using available online data does not exempt researchers from the ethical issues (Boyd & Crawford, 2012). They claim that simply because "the information was already on the web," it does not grant researchers the right to harvest large-scale information without undergoing proper procedures (Zimmer, 2010).

Our position is somewhere in-between. We want to conduct ethical research as well as advocate for established approaches for doing online research without tying our hands behind our back. Currently, there is no clear case with regard to the ownership of online data. Even the fight over the legality of web scraping has changed direction several times.[11] However, we should point out that the ownership issue is actually of little concern to us because though we collect information from websites, as social scientists, we have little interest in owning such data. We simply wish to perform analyses with it. Additionally, facts and statistics are not covered by copyright law (Mitchell, 2018). As a consequence, the three cases we showed in this article do not include the final data set, only the ways to produce the results.

We would also like to point out that the researchers are still bounded to ethics. There are several considerations for researchers when utilizing online data collection (Zimmer, 2010), such as challenges to the traditional nature of informed consent, properly identifying and respecting the privacy of websites, developing sufficient strategies for data anonymization, and addressing the importance of institutional review on human subjects. As a practitioner-oriented piece, the current study advocates the following practices.

For research purposes, we only scrap public data. By public we mean that any user can access the information without signing up. We strongly suggest analysts stay away from protected computers. We also instruct the analysts to show etiquette when scraping. This means typically checking for (a) the terms of service and (b) robots.txt because some sites allow scraping while many do not. For those that do, typically, when users sign up for the web service, they explicitly grant the website consent to use and reproduce their information for the website or for third parties. Therefore, it is acceptable for researchers to collect data via scraping following the site-permitted approaches.[12] We always recommend that the researcher contact the website before engaging in web scraping in order to obtain permission and clarify any data ownership issues. In many cases, permission is actually built into the API.

In another scenario, when there is no clear indication as to whether the website allows or prohibits scraping, we point readers' attention to the notion of *Fair Use* (Lessig, 2002). In essence, fair use is "Any copying of copyrighted material done for a limited and transformative purpose, such as to comment upon, criticize, or parody copyrighted work. Such uses can be done without permission from the copyright owner" (Stim, 2010, p. 244). The legal definition of "transformative" is sufficiently vague that researchers could use it to defend their own practice. In most cases, fair use analysis falls into the category of commentary/criticism and parody. Whether research falls into that category is open for debate, however.

One of the most important factors in deciding whether research falls into the fair use category depends on the amount and substantiality of the proportion taken. It is widely believed that the less one takes from the original content, the stronger the case for fair use. This fits well into the sampling framework of researchers. As academic researchers, we care less about individual particularities but rather seek to find regularity at the group level. As a result, having access to the entire population is ideal but not necessary. We could yield good enough estimates of the population given that we acknowledge the error in choosing the observed samples. For instance, a 5% sampling rate is vastly different from taking half of the population (Xin & Cai, 2019). Systematically querying a subset of the population is recommended once the researcher identifies their specific study population. Taking only a proportion of the entire population also helps to solve the issue of privacy and data anonymization, and using a random sample makes it much more difficult for a third party to reidentify the sampled individual.

During the data collection period, we advise analysts to stay identifiable even when using a preprogrammed `User-Agent` field (i.e., deploy robot crawlers). The analysts should insert identifiable information in the `User-Agent` field to let the website know who is accessing their data.[13] Users should also mind not to cause too much traffic and overload the site. This means not to scrape the same resources over and over. It also means to not put your own efficiency over another's browsing experience. We suggest limiting the interval with which analysts would send out http requests or headless browser requests (in our CJO example, we set our interval to 0.5–2 s per request).

Lastly, we highly recommend not posting data sets online because data ownership is a highly disputed area. An excellent legal discussion is offered by Bernard (2017) and more recently the Electronic Frontier Foundation (Fischer & Crocker, 2019). We encourage more readers to join the discussion of online data collection, as more data are being produced today than ever.

For institutional researchers, there is another consideration. In the cases where online data involve human subjects—and they mostly do in the social sciences—the institutional review boards or other forms of ethical review boards serve as the gatekeeper of human subject protection. However, depending on the type of data the analyst wishes to scrap, existing IRB rules may or may not be well equipped to protect subjects. Metcalf and Crawford (2016) pointed out that under the Common Rule principle, which is the primary regulation governing human subjects' research in the United States, projects that make use of publicly available or existing data sources are typically considered as posing minimal risk and thus qualify as exemptions. This is true in our cases. LAPD's news section and the CJO are both examples of publicly available government information. Twitter's own API puts restrictions on how much information we can get about users. However, it might be problematic in other cases, as even data that involve de-identified public data sets, when looked at as a whole, can produce unexpected harm (e.g., Metcalf & Crawford, 2016; Zimmer, 2010). As such, because there is no generally accepted practice or clearly delineated guidelines, analysts are left to self-regulate. We advocate for open and candid discussions with each institution's ethics board, as the interpretation of risks and harm ultimately puts the ethical burden on the analysts and the institution. From an individual analyst's point of view, to paraphrase Abbott (2016), we need to realize that this largely involuntary data generation process "does violence to the nature of these subjects" (p. 287). This is irreversible; thus, we as researchers much regularly remind ourselves to modify and monitor our practices, "not in the direction of making it 'scientific' or 'clean'," but for the purpose of making the research more humane.

Appendix

Glossary

- Anti-scraping mechanisms
  - Scraping web pages often involves a program sending a large number of requests to a server in a short period of time, which would increase the load to the server, potentially crashing it in certain cases. Thus, there are anti-scraping mechanisms on the server side to verify whether a request is initiated from a human client or a malicious program. If the request sender is determined to be a program, the server will simply refuse it.
- Application programming interface (API)
  - Most of the computer applications we use on a daily basis are designed as a graphic interface, which can be navigated easily by a mouse. But for software developers to achieve automation, they program various tasks. An API can satisfy developers with its programmable interfaces. Under an API, the developer can send proper control codes to the application, and the application would return the results to the developer.
- CAPTCHA
  - CAPTCHA is short for "Completely Automated Public Turing test to tell Computers and Humans Apart." This kind of strategy is broadly applied on contemporary websites (e.g., validation code on login web page). This strategy is an example of an anti-scraping mechanism. To avoid hostile or abnormal requests, the server provides the client with an image of twisted random characters, which can be recognized only by humans. Only

when the client recognizes and submits these twisted characters to the server, will the client pass the validation request, and one is allowed to get further information.

– CPU and RAM usage
  - Programs running on a computer consume both CPU (computing power) and RAM (memory) resources. Controlling a headless browser consumes a great amount of CPU and RAM resources as opposed to directly programming http requests.
– Developer mode of a web browser
  - Contemporary web browsers have integrated a system for web developers, which can inspect the HTML source code of the current browsing web page, and catch the details of requests and responses that a click on the web page triggers. On Google Chrome, one can simply right-click anywhere on the web page and select "Inspect" in the pop-up menu; on Firefox, one can select "Inspect Element."
– Firewall (see the China Judgement Online case)
  - A firewall in computer networks is a security measure for network communication. It identifies whether a connection is harmful to the protected system (e.g., high frequency of requesting). If a connection is identified as harmful, the firewall will directly reject the connection.
– Headless browser (maybe give an example? direct to an online tutorial?)
  - A web page browser without a graphical appearance. It is usually used by web developers for debugging their web applications. Since it can be programmed, we use it to replace human browsing activity in the CJO example. The pros and cons of a headless web page browser as compared to directly mimicking http requests are explained in the article.
  - Webdriver: Chrome and Firefox both have a headless mode, but there still needs to be a specific application—a webdriver—to control the headless web page browsers via programming. The webdriver for Chrome is called *chromedriver* (https://chromedriver.chromium.org/) and for Firefox is *geckodriver* (https://github.com/mozilla/geckodriver/releases).
– Hypertext markup language (HTML) file, which is a text-based web file.
  - HTML is the direct source code of a web page. Web page browsers directly generate a rendered web page from HTML source code. HTML has a nested structure identified with tags. In the LAPD example, the received data that we are dealing with are HTML source codes. We navigate through the tags, locate, and then extract the information we want from it.
  -
```
<html>
  <head>
        <title>A Webpage</title>
  </head>
  <body>
      <h1>The title</h1>
      <p>
         The paragraph.
      </p>
  </body>
</html>
```
– Hyper text transfer protocol (http)
  - HTTP, a commonly used protocol for client–server communication. The client sends an http request to the server, the server responds to the client with an http response. Both

request and response are formatted texts that consist of a `Header` part and a `Body` part. The `Body` contains the transmitted information, and the `Header` contains a description of the communication.

- Request and response: Usually pushed to the server side and from the server side
  - Both request and response will have a `Header` and `Body` component.
  - `User-Agent`: An information field in the header component of an http request that tells the server what is the initiator of the request. Most of the websites only respond to the requests initiated by a web browser. The following is an example:
  - `user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.90 Safari/537.36`
  - HTTP cookie:
- In the http communication between client and server, the headers of the requests and responses often include a field called "cookies," which preserves specific information to reference user-initiated communication behavior. The information inside the cookies field records the status of the communication, which reminds the server of the process of communication. In other circumstances, the validation information (e.g., login information) is recorded in this field to remind the server of the authority of the client. This is also an anti-scraping mechanism strategy.

– Internet protocol (IP) address
- Each node (a client or a server) of the Internet has an address to be located. An IP address consists of four numbers, each number values from 0 to 255.

```
192.168.0.1
```

– JavaScript
- A script programming language that is widely applied in contemporary web applications (e.g., websites). Due to the structure of a website, some programs should be executed on the client-side, which are usually JavaScript codes. The server usually sends a JavaScript source file (.js) to the client or embeds JavaScript codes inside HTML source codes. With JavaScript running inside the browser, a web page would no longer be a static HTML, and the browser can refresh a component of the web page instead of the whole page. JavaScript can also actualize the verification part of the web page (e.g., a user login) and the anti-scraping mechanism (e.g., generating web page content with program codes instead of statically embedding it in the HTML codes to puzzle the scraper).

– JavaScript object notation (JSON)
- This data format was born from JavaScript, but now it has been used much more broadly. Unlike the tagged structure of HTML and XML, JSON is organized as a "key: value" structure. JSON is often used to transmit small- scale data with frequent connections.
-
```
[
    {
        "name": "Alice",
        "gender": "female",
        "age": "29"
    },
    {
        "name": "Bob",
        "gender": "male",
        "age": "28"
    }
]
```

- Proxy:
  - Web servers typically limit the requests that are sent by the same client in a short period of time. The criterion to distinguish whether the requests are sent by the same client is based on IP address detection. A proxy is an IP address which is used to mask the actual location of a client's requests. It can be used to deploy multiple scraping programs simultaneously.
- Robot user
  - Most websites welcome human users while limiting the access of robot users (i.e., bots, crawlers, spiders). Robot users are automatic harvesters for different purposes. They are often equipped with preengineered http requests.
- Static and dynamic websites
  - A static site is where all the contents are pregenerated by the site administrator. The content on the site does not change according to user behavior.
- Uniform resource locator (URL)
  - URL, essentially an address. This address locates the path of a file in a computer file system, which can be local or remote. In this article, we assume the URL refers to the path of a web page. The following is the web page path of a URL's definition in Wikipedia: `https://en.wikipedia.org/wiki/URL`
- eXtensible markup language (XML)
  - Just like HTML, XML is also a tagged text file, the difference is that XML only contains the information but no layout description. It is often used to transmit large-scale structural data. Like HTML, data in XML format are supposed to be read and written by computer applications (e.g., web page browser).
  -
```
<quiz>
    <qanda seq="1">
        <question>
            Who was the forty-second president of the USA?
        </question>
        <answer>
            William Jefferson Clinton
        </answer>
    </qanda>
</quiz>
```

Authors' Note

The funding sources had no role in the design, analysis, interpretation, or reporting of results, or in the decision to submit the article for publication.

Data Availability

LAPD data can be found at: http://www.lapdonline.org
Sentencing documents can be retrieved from the CJO website at: http://wenshu.court.gov.cn
Tweets can be found at: https://twitter.com/search-advanced

Declaration of Conflicting Interests

The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The authors disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was supported by MYRG2018-00005-FED.

Software Information

All of the work was implemented in Python using libraries *Requests*, *Beautiful Soup*, *Twython* (Version No. 3.7.0), and the *Selenium* package. Please refer to the online repository for detailed environment setup.

Notes

1. We reviewed social science methodology textbooks from Sage, one of the major publishers. Among the 279 textbooks available, "web scraping" appeared 3 times (we performed the analysis by scraping the Table of Contents [ToC] of each title and examined physical copies of those not provided in the online ToC), which was typically found under specialized areas such as "text mining" or "Programming with Python/R." None of the comprehensive textbooks covered this topic, not a surprise since we found only 307 total entries under "web scraping" on the Web of Sciences, the majority of which belonged to computer science and engineering (roughly 244). In large disciplines such as statistics, economics, political science, sociology, and psychology (all subfields), there were only 32 entries from 1980 to 2019.
2. Paul Bauer wrote an excellent web book: https://bookdown.org/paul/big-data6/; William Marble's concise introduction is also very instructive: https://stanford.edu/□wpmarble/webscraping_tutorial/webscraping_tutorial.pdf.
3. For our purpose, we did not distinguish between the two terms "replication" and "reproducibility," which are both standard in the social sciences (Group A, see Barba, 2018). The major case (and its responses) are cited here, with the Open Science Collaboration fitting into this category. For an extended discussion, see National Academies of Sciences, Engineering, and Medicine (2019).
4. This is obviously not a precise classification. But for our purpose, we only distinguish web pages based on whether the content of each page changes after the user interact with it.
5. All the cases discussed in this article are implemented under Python and can be accessed here: https://github.com/f10w3r/trails_of_data. Because of the legal restrictions on sharing proprietary data publicly, we chose not to share the final data sets in the repository. These data sets are available upon request.
6. See the glossary in the Appendix for a description of the technical terms. We do not assume that readers must understand these terminologies in order to grasp the framework.
7. Therefore, in this article, we do not discuss any cases related to sites that require user registration to view. Twitter is perhaps the exception because although all the tweets are public to anyone on the Internet, an account is required to fetch data. But for other sites, the rules of engagement vary greatly depending on their terms of service. For simplicity, we focus on public websites here.
8. For a detailed explanation for social scientists, see Janetzko (2017).
9. This is from an engineering perspective and is perhaps more straightforward than other approaches, but because of ethical considerations, we advise analysts not to resort to this approach. Basically, for this approach to work, the analyst needs to manufacture a `User-Agent` field in the http requests with information that is likely not true in order to pass protection mechanisms.
10. The complication here is that Twitter changes the way it accepts advanced search requests from time to time. Given that this method is outside of the official-supported application programming interface (API), it is not very well documented. The analyst will have to monitor and analyze communication with the Twitter server to understand how to make this method work. It is not difficult to understand Twitter's motive for this, as it now also offers a premium version of its API (https://developer.twitter.com/en/premium-apis.html), which gives users similar functionality for a subscription fee.
11. The legal cases have changed from one direction to another just in a matter of years in the United States alone. See section 9.3 of Munzert, Rubba, Meissner, and Nyhuis (2015) for a recent review.

12. Appendix C of Mitchell (2018) offers an easy-to-follow introduction to this topic.
13. A good introduction on this topic can be found in section 9.3.3 of Munzert et al. (2015).

References

Abbott, A. (2016). *Processual sociology.* Chicago, IL: University of Chicago Press.

Anderson, C. J., Bahník, Š., Barnett-Cowan, M., Bosco, F. A., Chandler, J., Chartier, C. R., & Zuni, K. (2016). Response to comment on "estimating the reproducibility of psychological science." *Science*, *351*, 1037. doi: 10.1126/science.aad9163

Bainbridge, W. S. (2007). Computational sociology. In G. Ritzer (Ed.), *The Blackwell encyclopedia of sociology.* doi:10.1111/b.9781405124331.2007.x

Barba, L. A. (2018). Terminologies for reproducible research. *Computing Research Repository, abs/1802.03311* (arXiv:1802.03311).

Bernard, B. (2017, April 17). Web scraping and crawling are perfectly legal, right? [Blog post]. Retrieved from https://benbernardblog.com/web-scraping-and-crawling-are-perfectly-legal-right/

Blank, G. (2017). The digital divide among Twitter users and its implications for social research. *Social Science Computer Review*, *35*, 679–697. doi:10.1177/0894439316671698

Bonzanini, M. (2016). *Mastering social media mining with Python*. Birmingham, England: Packt.

Boyd, D., & Crawford, K. (2012). Critical questions for big data. *Information, Communication & Society*, *15*, 662–679.

Camerer, C. F., Dreber, A., Forsell, E., Ho, T. H., Huber, J., Johannesson, M., & Wu, H. (2016). Evaluating replicability of laboratory experiments in economics. *Science*, *351*, 1433–1436. doi:10.1126/science.aaf0918

Christensen, G. S., & Miguel, E. (2016, December). Transparency, reproducibility, and the credibility of economics research (Working Paper No. 22989). National Bureau of Economic Research. doi:10.3386/w22989

Crawford, K., Miltner, K., & Gray, M. L. (2014). Critiquing big data: Politics, ethics, epistemology. *International Journal of Communication*, *8*, 1663–1672.

Fischer, C., & Crocker, A. (2019, September 10). Victory! ruling in *hiQ v. Linkedin Protects Scraping of Public Data*. *Electronic Frontier Foundation*. Retrieved from https://www.eff.org/deeplinks/2019/09/victory-ruling-hiq-v-linkedin-protects-scraping-public-data

Frye, C., Plusch, M., & Lieberman, H. (2003). Static and dynamic semantics of the web. In W. Wahlster (Ed.), *Spinning the semantic web* (pp. 376–401). Cambridge, MA: MIT Press.

Gilbert, D. T., King, G., Pettigrew, S., & Wilson, T. D. (2016). Comment on "estimating the reproducibility of psychological science." *Science*, *351*, 1037–1037. doi:10.1126/science.aad7243

Herndon, J., & O'Reilly, R. (2016). Data sharing policies in social sciences academic journals: Evolving expectations of data sharing as a form of scholarly communication. In L. M. Kellem & K. Thompson (Eds.), *Databrarianship: The academic data librarian in theory and practice* (pp. 219–243). Chicago, IL: American Library Association.

Ignatow, G., & Mihalcea, R. (2017). *An introduction to text mining: research design, data collection, and analysis*. Thousand Oaks, CA: Sage.

Iliadis, A., & Russo, F. (2016). Critical data studies: An introduction. *Big Data & Society.* doi:10.1177/2053951716674238

Janetzko, D. (2017). The role of APIs in data sampling from social media. In L. Sloan & A. Quan-Haase (Eds.), *The Sage handbook of social media research methods* (*Chapter 10*). Thousand Oaks, CA: Sage.

King, G. (1995). Replication, replication. *Political Science & Politics*, *28*, 444–452. doi:10.2307/420301

Kouzis-Loukas, D. (2016). *Learning Scrapy.* Birmingham, England: Packt.

Landers, R. N., Brusso, R. C., Cavanaugh, K. J., & 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*, 475–492.

Lazer, D., & Radford, J. (2017). Data ex Machina: Introduction to big data. *Annual Review of Sociology*, *43*, 19–39. doi:10.1146/annurev-soc-060116-053457

Lessig, L. (2002). *The future of ideas: The fate of the commons in a connected world*. New York, NY: Vintage Press.

McFarland, D. A., Lewis, K., & Goldberg, A. (2016). Sociology in the era of big data: The ascent of forensic social science. *The American Sociologist*, *47*, 12–35.

McGrath, R. (2019). Twython (Version: 3.7.0): Pure Python wrapper for the Twitter API [Computer software]. Retrieved from https://github.com/ryanmcgrath/twython

Metcalf, J., & Crawford, K. (2016). Where are human subjects in big data research? The emerging ethics divide. *Big Data & Society*, *3*. doi:10.1177/2053951716650211

Mitchell, R. (2018). *Web scraping with Python: Collecting more data from the modern web* (2nd ed.). Sebastopol, CA: O'Reilly Media.

Moran, T. P. (2005). The sociology of teaching graduate statistics. *Teaching Sociology*, *33*, 263–284. doi:10.1177/0092055X0503300303

Munzert, S., Rubba, C., Meissner, P., & Nyhuis, D. (2015). *Automated data collection with R: A practical guide to web scraping and text mining*. Hoboken, NJ: Wiley.

Muthukadan, B. (2018). *Python: Selenium package [Computer software]*. Retrieved from https://selenium-python.readthedocs.io/

National Academies of Sciences, Engineering, and Medicine. (2019). *Reproducibility and replicability in science*. Washington, DC: The National Academies Press.

Peng, R. D. (2009). Reproducible research and biostatistics. *Biostatistics*, *10*, 405–408. doi:10.1093/biostatistics/kxp014

Peng, R. D. (2011). Reproducible research in computational science. *Science*, *334*, 1226–1227. doi:10.1126/science.1213847

Reitz, K. (2014). *Requests: Http for humans (Version 2.21.0) [Computer software]*. Retrieved from http://docs.python-requests.org/en/master/

Richardson, L. (2015). *Beautiful Soup (Version 4.6.3) [Computer software]*. Retrieved from https://www.crummy.com/software/BeautifulSoup/bs4/doc/#

Salganik, M. J. (2017). *Bit by bit: Social research in the digital age*. Princeton, NJ: Princeton University Press.

Stim, R. (2010). *Getting permission: How to license & clear copyrighted materials online & off*. Berkeley, CA: Nolo.

Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, *59*. Retrieved from http://www.jstatsoft.org/v59/i10/

Xin, Y., & Cai, T. (2019). Paying money for freedom: Effects of monetary compensation on sentencing for criminal traffic accident offenses in China. *Journal of Quantitative Criminology*. doi:10.1007/s10940-019-09409-w

Zimmer, M. (2010). "But the data is already public": On the ethics of research in Facebook. *Ethics and Information Technology*, *12*, 313–325. doi:10.1007/s10676-010-9227-5

Author Biographies

Fumin Li works in the Department of Sociology at University of Macau. E-mail: yb87313@connect.um.edu.mo

Yisu Zhou is associated with the Faculty of Education at University of Macau. E-mail: zhouyisu@um.edu.mo

Tianji Cai works in the Department of Sociology at University of Macau. E-mail: tjcai@um.edu.mo