Thank you for inviting me to be a part of this year’s summer institute in computational social science. I completed this very same institute organized by Vissého several years ago now and I cannot emphasize enough how much better a programmer I came out of that course. It really is an amazing experience and not only do you learn a lot of new things, but you get to make a lot of new friends and future collaborators as well.

So the subject of today’s training session is by far one of my favourite things to do as a researcher, which is web scraping. Web scraping as we are going to learn can be an incredibly powerful data collection technique to add to your toolkits and R has a lot of tools that make web scraping much easier than it often is in other programming languages.

There are a lot of tools available in R that we can use to do web scraping. Today I’m going to be teaching you about a package called rvest, but will mention several others as well closer to the end of the session. If you haven’t done any web scraping before, rvest is the place to start, and is the really most you’ll use the most to do web scraping in R. Other packages like RSelenium are important, but only necessary when we are dealing with more complicated scrapes of websites that make use of a lot JavaScript and dynamic content. If that was confusing, don’t worry, I’ll explain shortly what those things mean.

So what is web scraping. Web scraping is, in the simplest terms, the automated collection of information from the Internet. In the context of social science research, web scraping opens up many opportunities, but it also presents many challenges.

In terms of opportunities, web scraping provides access to new and accessible data that might not be available through other means. This could be a large text data set of pdf documents, a large corpus of tweets or reddit comments, a collection of news articles, or a data table of quantitative information collected from a wide variety of sources. Web scraping allows us to gather real-time information, which is essential for certain research projects. As I’ll mention at the end, there are now ways we can automate our R scripts to run every 10 minutes, every hour, or every week so that we are collecting information from the Internet over time. Web scraping enables us to conduct mixed-methods research, combining data from the Internet with other more traditional data sources like Census data or institutional data. Finally, web scraping opens up opportunities with respect to comparative research, by making it easier than ever to collecting data from multiple sources that may represent multiple time periods, countries, languages, and so on.

But of course there are many challenges that come with web scraping as well. Web scraping comes with ethical and legal considerations, and generally no easy answer to these. Most university ethics boards have yet to figure out what their position is on web scraping and the best practices for reviewing it. There are some best practices that we can follow. For example, it is essential to respect the guidelines provided by websites in their robots.txt files, which specify permissions and restrictions for scraping. I’ll talk more about robots.txt files in a moment. We can customize user-agents and headers to comply with website policies, essentially this is a way of notifying a webmaster that it is a researcher conducting the scrape and providing contact and other details about why we are doing it. We should be aware of the terms of use of the websites we scrape and adhere to any data usage restrictions. Finally, even if we feel our scrape is legally and ethically justifiable, there are going to be challenges related to the data itself – about quality, representation, and bias.

So let me say a bit more about Legal and Ethical Considerations. One best practice guideline I would recommend is to respect the guidelines outlined in the robots.txt file of a website. I’m not going to say you have to follow these, as most of the time they were not written with academic researchers in mind, but you should always start by inspecting it. The robots.txt file specifies what can and cannot be accessed by web crawlers, including scraping bots. It provides insights into the website's policies regarding data collection and scraping activities. Later in the session I’m going to teach you how to read these files and make sense of them.

Okay, so we know this robots.txt file provides guidelines on what we can scrape form a website and how we should design our scraper. How do we find it? The robots.txt file is a text file located at the root of a website that provides instructions to web crawlers, including scraping bots, on what can and cannot be accessed.

There’s some jargon we need to understand. Firstly, "User-agent". This refers to the specific bot or crawler to which the following directives apply. Different bots may have different rules and permissions specified for them.

Below user-agent, we usually find "Allow" and "Disallow" directives. These indicate which parts of the website are allowed or disallowed to be accessed by the specified user-agent. The "Allow" directive lists the URLs or directories that our scraper is allowed to extract from, while the "Disallow" directive lists those that should be considered off limits from our scrape.

Finally, you’ll find a "Crawl-delay" in some robots.txt files. This is an optional directive that specifies the time delay, in seconds, that our scraping algorithm or bot should wait between successive requests to the website. It’s used to manage strain on the server. Too many requests, from too many computers, too quickly can cause the server to crash, which is bad. In fact, hackers will do this purposefully in what’s known as a denial of service attack. We don’t want to do this.

Okay so let’s say we’ve identified a website we want to scrape, we’ve inspected the robots.txt and maybe also the terms and service of use. Now we want to conduct a scrape. How do we do it?

Before we can get there, we have to know a little bit about the Internet and the building blocks used to create what we call websites. Most websites are built on three core programming languages: HTML, CSS, and JavaScript. We don’t need to be proficient in these programming languages to be able to do web scraping in R, but we do have to know some basics.

I’m going to start with HTML. HTML stands for (HyperText Markup Language). It’s is a language used to structure and present content on the web. It uses tags to define HTML elements such as headings, paragraphs, lists, etc. These elements can be nested inside each other to create a hierarchical structure. Even if you’ve never written any HTML before, I’m sure you’ll recognize it.

Web scraping involves extracting content from the “source code” of a website. So when we are designing our scraper, we want to inspect this source code so we can see how it’s structured and come up with a plan to locating and extracting the content we want. To access a website’s source code, we can use the browser's developer tools. Every browser has a developer mode. By right-clicking on a web page and selecting "Inspect," we can access the developer tools. On macOS, the shortcut is ⌘ + Option + I, and on PC, it's Ctrl + Shift + I.

An HTML document is composed of several essential parts that contribute to its overall structure and functionality. Firstly, the <html> element serves as the root element of the HTML page, encapsulating all other elements within it. The <head> element contains metadata about the document, such as the title of the page, links to stylesheets or scripts, and other important information that is not directly displayed on the webpage. The <body> element represents the main content of the HTML page that is visible to the users. It includes text, images, videos, links, and other elements that make up the actual content displayed in the browser. These three elements work together to define the structure and presentation of an HTML document.

Within these three levels of the HTML document, what we call the Document Object Model or DOM, which represents the hierarchical structure of an HTML document, we find a wide array of nested elements that contribute to the content and layout of the webpage. For instance, <p> tags are used to define paragraphs of text, allowing for organized and structured presentation of textual information. <a> tags create hyperlinks, enabling users to navigate to other web pages or sections within the same page. Additionally, <div> elements are commonly used as containers to group and organize other elements, providing flexibility in styling and layout. These nested elements, along with many others such as headings (<h1>-<h6>), lists (<ul>, <ol>), images (<img>), and more, allow developers to create intricate and visually appealing webpages.

When it comes to working with HTML tags, one challenge is their inherent generality. HTML tags, such as <div>, <p>, or <span>, serve as building blocks for structuring content on a webpage. However, these tags alone do not provide enough specificity to target and extract the precise information we need during web scraping. This is where CSS selectors come into play. By utilizing CSS selectors, we can specify criteria beyond the generic tag names to precisely identify and extract the desired elements. CSS selectors allow us to target elements based on various attributes like class names, IDs, or even their hierarchical relationships within the HTML structure. This added specificity offered by CSS selectors empowers us to effectively navigate and extract the relevant data from HTML documents during web scraping.

CSS selectors are patterns used to select elements based on their tag names, class names, IDs, attributes, and hierarchical relationships. By employing CSS selectors, we can extract relevant information from web pages during scraping. For instance, we can use tag selectors (e.g., <div>, <p>) to extract specific types of elements. Class selectors (e.g., .class-name) allow us to target elements with specific class attributes, while ID selectors (e.g., #id-name) help us extract elements with unique IDs. Attribute selectors (e.g., [attribute=value]) allow for extraction based on specific attribute values. Additionally, CSS selectors provide powerful techniques such as descendant selectors, child selectors, and sibling selectors, which facilitate targeting elements based on their hierarchical relationships within the HTML structure.

Two important CSS selectors to be aware of for web scraping are classes, IDs, and attribute selectors. Classes in CSS are denoted by a period (.), followed by the class name. They allow you to target elements that belong to a specific class. For example, if an element has a class attribute of "my-class", you can select it using the class selector as ".my-class". This is useful when multiple elements share the same styling or functionality.

IDs in CSS are identified by a hash symbol (#), followed by the ID name. IDs are unique to each element on a page, and they provide a way to target a specific element directly. For instance, if an element has an ID attribute of "my-id", you can select it using the ID selector as "#my-id". IDs are commonly used when there is a need to style or interact with a particular element uniquely.

Another approach for extracting information from HTML is by using XPath. XPath is a language used to navigate through HTML documents. With XPath, we can define paths and patterns to select elements based on their location in the document tree. XPath uses expressions to identify elements by their tag names, attributes, relationships, and more. It provides flexibility in targeting specific elements regardless of their class names or IDs. XPath expressions can traverse the document hierarchy, allowing for precise element selection by specifying parent-child relationships, sibling relationships, or even more complex conditions.

I want to take a moment here to mention a really useful tool called SelectorGadget. SelectorGadget is a handy tool that helps us identify and automatically generate CSS selectors and XPaths for specific HTML elements. It simplifies the process of selecting and extracting data from web pages.

Okay so once we have the URL for the page we want to scrape, we’ve inspected the HTML using developer mode, we’ve identified CSS selectors or XPaths that will get us the content we want, it’s time to extract the information. In most cases, we can use a package in R called rvest, which contains tools for getting the HTML into our R environments, parsing it, and extracting the parts we want.

Okay so before we take a short break and then turn to the tutorial, which is the fun part, I want comment on two more areas of web scraping. First, I want to say, rvest is not always going to be enough.

While rvest is a powerful and popular web scraping package in R, it primarily targets static web content, meaning it is designed to extract data from web pages that do not change dynamically. Static web content refers to web pages where the HTML structure and data remain constant over time.

However, there are instances where web scraping requires extracting data from dynamic web pages. Dynamic web content, on the other hand, refers to web pages that incorporate elements that change dynamically, often through JavaScript or other scripting languages. These elements could include dynamically loaded data, interactive features, or real-time updates.

To handle dynamic web scraping scenarios, rvest alone may not suffice. In such cases, an alternative approach is to employ tools like RSelenium or utilize APIs (Application Programming Interfaces).

RSelenium is an R package that provides a way to automate web browsers and interact with dynamic web content. It enables the scraping of web pages that heavily rely on JavaScript by controlling a real browser programmatically, usually called a remote browser. By automating browser actions, RSelenium allows you to extract data from web pages that are rendered dynamically.

Alternatively, many websites provide APIs that allow developers to access and retrieve specific data in a structured manner. Sometimes these are really expensive, Twitter’s API now costs $45,000 USD for researchers, while others can be free. APIs are a set of rules and protocols that enable different software applications to communicate with each other, in this case our computer with the backend server of a webpage, allowing us to request and obtain the information we want.

When working with APIs, you may need to obtain an API key, which acts as a unique identifier for your application and grants you access to the API's resources. This key is typically provided by the service hosting the API and ensures that only authorized users can make requests and retrieve data. Sometimes APIs are client-side, and I’ll briefly get into what this means at the end

Finally, I want to mention that it’s possible, and now much easier than it was in the past, to run our R scrapes on a remote server on a scheduler. For example, we can write scrapes that run every 10 minutes, or every hour, every day, every week, and pool the data into a spread hosted somewhere in the cloud, such as on google sheets. Platforms like GitHub Actions, AWS, and Heroku provide cloud-based web scraping solutions. This is pretty advanced in the web scraping world, but with practice you can definitely get there in a matter of days or weeks, and there’s a lot of really good tutorials online that walk you through how to do it.