**Introduction**

Whenever a new paper is released using some type of scraped data, most of my peers in the social science community get baffled at how researchers can do this. In fact, many social scientists can’t even think of research questions that can be addressed with this type of data simply because they don’t know it’s even possible. As the old saying goes, when you have a hammer, every problem looks like a nail.

With the increasing amount of data being collected on a daily basis, it is eminent that scientists start getting familiar with new technologies that can help answer old questions. Moreover, we need to be adventurous about cutting edge data sources as they can also allow us to ask new questions which weren’t even thought of in the past.

In this tutorial I’ll be guiding you through the basics of web scraping using R and the xml2 package. I’ll begin with a simple example using fake data and elaborate further by trying to scrape the location of a sample of schools in Spain.

**Basic steps**

For web scraping in R, you can fulfill almost all of your needs with the xml2 package. As you wander through the web, you’ll see many examples using the rvest package. xml2 and rvest are very similar so don’t feel you’re lacking behind for learning one and not the other. In addition to these two packages, we’ll need some other libraries for plotting locations on a map (ggplot2, sf, rnaturalearth), identifying who we are when we scrape (httr) and wrangling data (tidyverse).

Additionally, we’ll also need the package scrapex. In the real-world example that we’ll be doing below, we’ll be scraping data from the website [www.buscocolegio.com](http://www.buscocolegio.com) to locate a sample of schools in Spain. However, throughout the tutorial we won’t be scraping the data directly from their real-website. What would happen to this tutorial if 6 months from now [www.buscocolegio.com](http://www.buscocolegio.com) updates the design of their website? Everything from our real-world example would be lost.

Web scraping tutorials are usually very unstable precisely because of this. To circumvent that problem, I’ve saved a random sample of websites from some schools in [www.buscocolegio.com](http://www.buscocolegio.com) into an R package called scrapex. Although the links we’ll be working on will be hosted locally on your machine, the HTML of the website should be very similar to the one hosted on the website (with the exception of some images/icons which were deleted on purpose to make the package lightweight).

You can install the package with:

# library("scrapex")

Now, let’s move on the fake data example and load all of our packages with:

library(xml2)

library(httr)

library(tidyverse)

library(sf)

library(rnaturalearth)

library(ggplot2)

library(scrapex)

Let’s begin with a simple example. Below we define an XML string and look at its structure:

xml\_test <- "

Jason

Bourne

Spy

Carol

Kalp

Scientist

"

cat(xml\_test)

##

##

##

##

##

## Jason

##

##

##

## Bourne

##

##

## Spy

##

##

##

##

##

##

##

## Carol

##

##

##

## Kalp

##

##

## Scientist

##

##

##

##

In XML and HTML the basic building blocks are something called tags. For example, the first tag in the structure shown above is . This tag is matched by at the end of the string:

If you pay close attention, you’ll see that **each** tag in the XML structure has a beginning (signaled by <>) and an end (signaled by ). For example, the next tag after is and right before the tag is the end of the jason tag .

Similarly, you’ll find that the tag is also matched by a finishing tag.

In theory, tags can have whatever meaning you attach to them (such as or ). However, in practice there are hundreds of tags which are standard in websites. If you’re just getting started, there’s no need for you to learn them but as you progress in web scraping, you’ll start to recognize them (one brief example is which simply **bolds** text in a website).

The xml2 package was designed to read XML strings and to navigate the tree structure to extract information. For example, let’s read in the XML data from our fake example and look at its general structure:

xml\_raw <- read\_xml(xml\_test)

xml\_structure(xml\_raw)

##

##

##

##

##

## {text}

##

## {text}

##

## {text}

##

##

##

##

## {text}

##

## {text}

##

## {text}

You can see that the structure is tree-based, meaning that tags such as and are nested within the tag. In XML jargon, is the **root node**, whereas and are the **child nodes** from .

In more detail, the structure is as follows:

* The **root** node is
* The **child** nodes are and
* Then each **child** node has nodes , , and nested within them.

Put another way, if something is nested within a **node**, then the nested node is a **child** of the upper-level node. In our example, the **root** node is so we can check which are its children:

# xml\_child returns only one child (specified in search)

# Here, jason is the first child

xml\_child(xml\_raw, search = 1)

## {xml\_node}

##

## [1] \n \n \n Ja ...

# Here, carol is the second child

xml\_child(xml\_raw, search = 2)

## {xml\_node}

##

## [1] \n \n \n Carol\n ...

# Use xml\_children to extract \*\*all\*\* children

child\_xml <- xml\_children(xml\_raw)

child\_xml

## {xml\_nodeset (2)}

## [1] \n \n \n \n \n \n \n ...

Tags can also have different attributes which are usually specified as and ended as usual with . If you look at the XML structure of our example, you’ll notice that each tag has an attribute called type. As you’ll see in our real-world example, extracting these attributes is often the aim of our scraping adventure. Using xml2, we can extract all attributes that match a specific name with xml\_attrs.

# Extract the attribute type from all nodes

xml\_attrs(child\_xml, "type")

## [[1]]

## named character(0)

##

## [[2]]

## named character(0)

Wait, why didn’t this work? Well, if you look at the output of child\_xml, we have two nodes on which are for and .

child\_xml

## {xml\_nodeset (2)}

## [1] \n \n \n \n \n \n \n ...

Do these tags have an attribute? No, because if they did, they would have something like . What we need is to look down at the tag within and and extract the attribute from .

Does this sound familiar? Both and have an associated tag below them, making them their children. We can just go down one level by running xml\_children on these tags and extract them.

# We go down one level of children

person\_nodes <- xml\_children(child\_xml)

# is now the main node, so we can extract attributes

person\_nodes

## {xml\_nodeset (2)}

## [1] \n \n \n Ja ...

## [2] \n \n \n Carol\n ...

# Both type attributes

xml\_attrs(person\_nodes, "type")

## [[1]]

## type

## "fictional"

##

## [[2]]

## type

## "real"

Using the xml\_path function you can even find the ‘address’ of these nodes to retrieve specific tags without having to write down xml\_children many times. For example:

# Specific address of each person tag for the whole xml tree

# only using the `person\_nodes`

xml\_path(person\_nodes)

## [1] "/people/jason/person" "/people/carol/person"

We have the ‘address’ of specific tags in the tree but how do we extract them automatically? To extract specific ‘addresses’ of this XML tree, the main function we’ll use is xml\_find\_all. This function accepts the XML tree and an ‘address’ string. We can use very simple strings, such as the one given by xml\_path:

# You can use results from xml\_path like directories

xml\_find\_all(xml\_raw, "/people/jason/person")

## {xml\_nodeset (1)}

## [1] \n \n \n Ja ...

The expression above is asking for the node "/people/jason/person". This will return the same as saying xml\_raw %>% xml\_child(search = 1). For deeply nested trees, xml\_find\_all will be many times much cleaner than calling xml\_child recursively many times.

However, in most cases the ‘addresses’ used in xml\_find\_all come from a separate language called XPath (in fact, the ‘address’ we’ve been looking at **is** XPath). XPath is a complex language (such as regular expressions for strings) which is beyond this brief tutorial. However, with the examples we’ve seen so far, we can use some basic XPath which we’ll need later on.

To extract all the tags in a document, we can use //name\_of\_tag.

# Search for all 'married' nodes

xml\_find\_all(xml\_raw, "//married")

## {xml\_nodeset (2)}

## [1] \n Jason\n

## [2] \n Carol\n

With the previous XPath, we’re searching for **all** married tags within the complete XML tree. The result returns all married nodes (I use the words tags and nodes interchangeably) in the complete tree structure. Another example would be finding all tags:

xml\_find\_all(xml\_raw, "//occupation")

## {xml\_nodeset (2)}

## [1] \n Spy\n

## [2] \n Scientist\n

If you want to find any other tag you can replace "//occupation" with your tag of interest and xml\_find\_all will find all of them.

If you wanted to find all tags **below** your current node, you only need to add a . at the beginning: ".//occupation". For example, if we dived into the tag and we wanted his tag, "//occupation" will returns **all** tags. Instead, ".//occupation" will return only the found tags **below** the current tag. For example:

xml\_raw %>%

# Dive only into Jason's tag

xml\_child(search = 1) %>%

xml\_find\_all(".//occupation")

## {xml\_nodeset (1)}

## [1] \n Spy\n

# Instead, the wrong way would have been:

xml\_raw %>%

# Dive only into Jason's tag

xml\_child(search = 1) %>%

# Here we get both occupation tags

xml\_find\_all("//occupation")

## {xml\_nodeset (2)}

## [1] \n Spy\n

## [2] \n Scientist\n

The first example only returns ’s occupation whereas the second returned **all** occupations, regardless of where you are in the tree.

XPath also allows you to identify tags that contain only one specific **attribute**, such as the one’s we saw earlier. For example, to filter all tags with the attribute filter set to fictional, we could do it with:

# Give me all the tags 'person' that have an attribute type='fictional'

xml\_raw %>%

xml\_find\_all("//person[@type='fictional']")

## {xml\_nodeset (1)}

## [1] \n \n \n Ja ...

If you wanted to do the same but for the tags **below** your current nodes, the same trick we learned earlier would work: ".//person[@type='fictional']". These are just some primers that can help you jump easily to using XPath, but I encourage you to look at other examples on the web, as complex websites often require complex XPath expressions.

Before we begin our real-word example, you might be asking yourself how you can actually **extract** the text/numeric data from these **nodes**. Well, that’s easy: xml\_text.

xml\_raw %>%

xml\_find\_all(".//occupation") %>%

xml\_text()

## [1] "\n Spy\n " "\n Scientist\n "

Once you’ve narrowed down your tree-based search to one single piece of text or numbers, xml\_text() will extract that for you (there’s also xml\_double and xml\_integer for extracting numbers). As I said, XPath is really a huge language. If you’re interested, [this](https://devhints.io/xpath) XPath cheat sheets have helped me a lot to learn tricks for easy scraping.

**Real-world example**

We’re interested in making a list of many schools in Spain and visualizing their location. This can be useful for many things such as matching population density of children across different regions to school locations. The website [www.buscocolegio.com](http://www.buscocolegio.com) contains a database of schools similar to what we’re looking for. As described at the beginning, instead we’re going to use scrapex which has the function spanish\_schools\_ex() containing the links to a sample of websites from different schools saved locally on your computer.

Let’s look at an example for one school.

school\_links <- spanish\_schools\_ex()

# Keep only the HTML file of one particular school.

school\_url <- school\_links[13]

school\_url

## [1] "/usr/local/lib/R/site-library/scrapex/extdata/spanish\_schools\_ex/school\_3006839.html"

If you’re interested in looking at the website interactively in your browser, you can do it with browseURL(prep\_browser(school\_url)). Let’s read the HTML (XML and HTML are **usually** interchangeable, so here we use read\_html).

# Here we use `read\_html` because `read\_xml` is throwing an error

# when attempting to read. However, everything we've discussed

# should be the same.

school\_raw <- read\_html(school\_url) %>% xml\_child()

school\_raw

## {html\_node}

##