Stat 8054 Lecture Notes: Web Scraping

Charles J. Geyer

March 01, 2023

1 License

2 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.20.
- The version of the `rvest` package used to make this document is 1.0.3.
- The version of the `xml2` package used to make this document is 1.3.3.
- The version of the `rdom` package used to make this document is 0.0.3.9001.

Package `rdom` is not on CRAN but on Github. Install via

```
library("remotes")
install_github("https://github.com/cpsievert/rdom/")
```

3 Reading

- XPATH Tutorial from w3schools.com

4 View Source

To computers web pages look like what we see when we "view source". The accelerator Ctrl-U (control u) shows this in Firefox, Chrome, Opera, and Safari (flower U on Apple, of course). I don't know if there is an accelerator for Microsoft Edge. You will just have to mouse around the menus to find it.

To understand this material you have to stop thinking like a human and start thinking like a computer. Web pages are really "view source" not what browsers show your eyes.

5 Scraping Data from Tables

The job of data scraping (getting data from web pages) is easiest when the data

- are in an HTML table (surrounded by HTML elements <TABLE> and </TABLE>) and
- the table does not use any stupid tricks for visual look – it is a pure data table. Another way to say this is that all of the presentation is in CSS; the HTML is pure data structure.

Then CRAN package `rvest` grabs tables and puts the data in R data frame.

```
library(rvest)
u <- "https://www.ncaa.com/rankings/volleyball-women/d1/ncaa-womens-volleyball-rpi/"
foo <- read_html(u) |> html_element("table") |> html_table()
class(foo)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(foo)
```

```
## [1] 344   8
```

```
head(foo)
```

```
## # A tibble: 6 x 8
##    Rank School     Conference Record Road  Neutral Home  `Non Div I`
##   <int> <chr>      <chr>      <chr>  <chr> <chr>   <chr> <chr>
## 1     1 Texas      Big 12     22-1   9-1   0-0     13-0  0-0
## 2     2 Louisville ACC        26-2   12-1  2-0     12-1  0-0
## 3     3 Pittsburgh ACC        27-3   11-1  3-1     13-1  0-0
## 4     4 Wisconsin  Big Ten    25-3   11-1  1-1     13-1  0-0
## 5     5 Stanford   Pac-12     24-4   14-1  0-1     10-2  0-0
## 6     6 San Diego  WCC        27-1   10-1  3-0     14-0  0-0
```

6 Scraping Data from HTML

6.1 HTML

The job of data scraping is harder when the data not in one or more HTML tables but rather in unstructured or CSS structured HTML. By "CSS structured" we mean that the data are not just in plain HTML elements but rather in HTML elements that have classes defined by the programmers so we see things like

```
<FOO CLASS="wadget">this is the data</FOO>
```

where "FOO" is not the name of a valid HTML element, but is to be replaced by the name of any valid HTML element, for example,

```
<DIV CLASS="wadget">this is the data</DIV>
```

HTML element names and attribute names are not case sensitive, so the preceding example is the same as

```
<div class="wadget">this is the data</div>
```

but HTML class names, like `wadget` here are case sensitive, so

```
<div class="Wadget">this is the data</div>
```

defines a different class.

6.2 NCAA Women's Volleyball Tournament

The data we are going to get are the 2021 NCAA Division I Women's Volleyball Tournament. The URL is

```
u <- "https://www.ncaa.com/brackets/volleyball-women/d1/2021"
```

Now if we look at one at the source code for this page ("view source"), we see that the whole bracket is in a `div` element having attribute 'class="vue bracket-wrapper" (at least I hope so, that they are doing the same thing now as in 2018 when I first did this)

```
library("xml2")
foo <- read_html(u)
foo
```

```
## {html_document}
## <html lang="en" dir="ltr" prefix="og: https://ogp.me/ns#">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body class="path-taxonomy">\n        <a href="#main-content" class="visu ...
```

```
bar <- xml_find_all(foo, "//div[@class='vue bracket-wrapper']")
bar
```

```
## {xml_nodeset (1)}
## [1] <div class="vue bracket-wrapper">\n        <!-- Drupal integration templa ...
```

Looking further, we see that within there, every match seems to be in a `div` element with attribute `class="teams"`. Let's get them.

```
baz <- xml_find_all(bar, ".//div[@class='teams']")
baz
```

```
## {xml_nodeset (63)}
##  [1] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
##  [2] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
##  [3] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
##  [4] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
##  [5] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
##  [6] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
##  [7] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
##  [8] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
##  [9] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
## [10] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
## [11] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
## [12] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
## [13] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
## [14] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
## [15] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
## [16] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
## [17] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
## [18] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
## [19] <div class="teams" data-v-110394ce>\n<div class="team" data-v-072edef4 d ...
## [20] <div class="teams" data-v-110394ce>\n<div class="team winner" data-v-072 ...
## ...
```

```
length(baz)
```

```
## [1] 63
```

That's right. In a 64-team single elimination tournament there are 63 matches.

What is the dot in front of the XPATH argument? The example on the help page for function `xml_find_all` mentions it, but is not clear. Similarly for the XPATH Tutorial.

The way I think of it (which may not be pedantically correct) is that the value returned by R function `xml_find_all` is an XML nodeset (an R vector of XML nodes – at least nodes can be selected using the R single square brackets operator), but unlike what you may think (what I thought when I was confused about this issue) this is a nodeset within the original document. R object `bar` is not a cut down substructure of R object `foo`. Rather it is R object `foo` plus pointers to the nodes in the nodeset. So an XPATH that starts

"at the top" means at the top of the original document. We need the dot operator to say we want to start at the node that is R object `bar`. (More on this later when we use the dot-dot operator.)

So what is in one of these?

```
qux <- xml_find_all(baz[1], ".//div")
qux
```

```
## {xml_nodeset (2)}
## [1] <div class="team winner" data-v-072edef4 data-v-110394ce>\n<img src="//i. ...
## [2] <div class="team" data-v-072edef4 data-v-110394ce>\n<img src="//i.turner. ...
```

There are only two `div` elements in there. One for each team in the match. We can get the names out right away. They seem to be in a `span` element with attribute `class="name="`.

```
qux <- xml_find_all(baz, ".//div[@class='team winner']//span[@class='name']")
winner <- xml_text(qux)
qux <- xml_find_all(baz, ".//div[@class='team']//span[@class='name']")
loser <- xml_text(qux)
```

Great! We have gotten at least some data out of this web page! Also we have implicit information about what round each of these matches was in.

- Any team that does not appear in the `winner` vector lost in the first round.
- Any team that appears exactly once in the `winner` vector lost in the second round.
- Any team that appears exactly twice in the `winner` vector lost in the third round (sweet sixteen).
- Any team that appears exactly three times in the `winner` vector lost in the fourth round (elite eight).
- Any team that appears exactly four times in the `winner` vector lost in the fifth round (final four).
- Any team that appears exactly five times in the `winner` vector lost in the sixth round (national championship game).
- Any team that appears exactly six times in the `winner` vector is the national champion, the only team that does not appear in the `loser` vector.

That national champion is

```
setdiff(winner, loser)
```

```
## [1] "Wisconsin"
```

```
sum(winner == setdiff(winner, loser))
```

```
## [1] 6
```

We can also get seed information from this web page.

```
qux <- xml_find_all(baz, ".//div[@class='team winner']//span[@class='seed']")
winner.seed <- xml_text(qux)
winner.seed <- suppressWarnings(as.numeric(winner.seed))
qux <- xml_find_all(baz, ".//div[@class='team']//span[@class='seed']")
loser.seed <- xml_text(qux)
loser.seed <- suppressWarnings(as.numeric(loser.seed))
```

Only 16 teams are seeded. Check that

```
sort(unique(winner.seed))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

```
sort(unique(loser.seed))
```

```
##  [1]  1  2  3  5  6  7  8  9 10 11 12 13 14 15 16
```

Clearly the number 4 seed had no losses (so was the national champion).

7 Scraping Data from HTML Generated by JavaScript

This section no longer works. Don't know why. The best solution to this problem is RSelenium, but that is very complicated to use, and I have not tried it myself.

JavaScript (also called ECMAScript) is the world's most popular programming language. Every web browser is a JavaScript engine. Most web pages use JavaScript to do something on the web page. The web pages for my courses are very unusual nowadays in using no JavaScript.

Some web pages use JavaScript to create the data you see when you load the page in a browser. As an example, if you click on any of the boxes containing matches in the NCAA tournament bracket web page, you go to a web page having more information on that match, but if you view source for that page, you do not see any of that data. How can we get that data?

First let us get those URL's.

```
qux <- xml_find_all(baz, "..")
qux
```

```
## {xml_nodeset (63)}
##  [1] <a href="/game/5909401" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [2] <a href="/game/5909402" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [3] <a href="/game/5909403" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [4] <a href="/game/5909404" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [5] <a href="/game/5909405" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [6] <a href="/game/5909406" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [7] <a href="/game/5909407" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [8] <a href="/game/5909408" data-v-110394ce><div class="pod-status" data-v-1 ...
##  [9] <a href="/game/5909433" data-v-110394ce><div class="pod-status" data-v-1 ...
## [10] <a href="/game/5909434" data-v-110394ce><div class="pod-status" data-v-1 ...
## [11] <a href="/game/5909435" data-v-110394ce><div class="pod-status" data-v-1 ...
## [12] <a href="/game/5909436" data-v-110394ce><div class="pod-status" data-v-1 ...
## [13] <a href="/game/5909449" data-v-110394ce><div class="pod-status" data-v-1 ...
## [14] <a href="/game/5909450" data-v-110394ce><div class="pod-status" data-v-1 ...
## [15] <a href="/game/5909457" data-v-110394ce><div class="pod-status" data-v-1 ...
## [16] <a href="/game/5909417" data-v-110394ce><div class="pod-status" data-v-1 ...
## [17] <a href="/game/5909418" data-v-110394ce><div class="pod-status" data-v-1 ...
## [18] <a href="/game/5909419" data-v-110394ce><div class="pod-status" data-v-1 ...
## [19] <a href="/game/5909420" data-v-110394ce><div class="pod-status" data-v-1 ...
## [20] <a href="/game/5909421" data-v-110394ce><div class="pod-status" data-v-1 ...
## ...
```

```
urls <- xml_attr(qux, "href")
head(urls)
```

```
## [1] "/game/5909401" "/game/5909402" "/game/5909403" "/game/5909404"
## [5] "/game/5909405" "/game/5909406"
```

```
urls <- url_absolute(urls, u)
head(urls)
```

```
## [1] "https://www.ncaa.com/game/5909401" "https://www.ncaa.com/game/5909402"
```

```
## [3] "https://www.ncaa.com/game/5909403" "https://www.ncaa.com/game/5909404"
## [5] "https://www.ncaa.com/game/5909405" "https://www.ncaa.com/game/5909406"
```

Here our XPATH argument started with the dot-dot operator (`..`), which (XPATH Tutorial) selects the parent of the current node. This shows us that R object `baz` does not just contain the information of the XML nodeset that it (supposedly) is. Rather it contains the information of the whole XML document (the same as R object `foo`) plus the additional information indicating the nodeset. The XPATH `".."` says we want the HTML element that encloses each of the nodes in the nodeset `baz`.

We can grab the HTML generated by the JavaScript on one of those pages using R package `rdom` (found on GitHub not CRAN but discussed in the CRAN Task View on Web Technologies and Services). Install as described in Section 2 above.

```
library("rdom")
invisible(rdom(urls[1], filename = "foo.html"))
```

Now if we look at that file, we will see the HTML that the web browser generates using JavaScript. Now that we have it, working with it is just like working with HTML not generated by JavaScript.

We will only do one example, the locations of the matches.

```
fred <- read_html("foo.html")
sally <- xml_find_all(fred, "//span[@class='venue']")
xml_text(sally)
```

```
## character(0)
```

So lets try that on all the URL's.

```
doit <- function(url) {
    rdom(url, filename = "foo.html")
    fred <- read_html("foo.html")
    sally <- xml_find_all(fred, "//span[@class='venue']")
    xml_text(sally)
}

locs <- sapply(urls, doit)
head(locs)
```

It is a bit annoying that we have to download the file and cannot just keep it in R. The reasons is that R functions `rdom` and `read_html` do not use the same format. The documentation for rdom suggests using R package `rvest` instead of `xml2` to get around this. But so small a proportion of the time is spent writing and reading file `foo.html` that there is really no reason to worry about this.

We could have also used a filename created by R function `tempfile` if we wanted to avoid any possibility of clobbering a file in the current working directory. Here we used the name `"foo.html"` because we needed to look at it to see how to get data out of it.