Data Science for Economists

Lecture 6: Webscraping: (1) Server-side and CSS

Grant R. McDermott

University of Oregon | EC 607

Contents

Software requirements

External software

Today we'll be using SelectorGadget, which is a Chrome extension that makes it easy to discover CSS selectors. (Install the extension directly here.) Please note that SelectorGadget is only available for Chrome. If you prefer using Firefox, then you can try ScrapeMate.

R packages

- New: **rvest**, **janitor**
- Already used: **tidyverse**, **lubridate**, **data.table**, **hrbrthemes**

Recall that **rvest** was automatically installed with the rest of the tidyverse. However, these lecture notes assume that you have **rvest** 1.0.0, which — at the time of writing — has to installed as the development version from GitHub. The code chunk below should take care of installing (if necessary) and loading the packages that you need for today's lecture.

```
## Install development version of rvest if necessary
if (numeric_version(packageVersion("rvest")) < numeric_version('0.99.0')) {
  remotes::install_github('tidyverse/rvest')
}
## Load and install the packages that we'll be using today
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse, rvest, lubridate, janitor, data.table, hrbrthemes)
## My preferred ggplot2 plotting theme (optional)
theme_set(hrbrthemes::theme_ipsum())
```

> **Tip:** If you can get an error about missing fonts whilst following along with this lecture, that's probably because you don't have Arial Narrow — required by the `hrbrthemes::theme_ipsum()` **ggplot2** theme that I'm using here — installed on your system. You can resolve this by downloading the font and adding it to your font book (Google it), or by switching to a different theme (e.g. `theme_set(theme_minimal())`).

Webscraping basics

The next two lectures are about getting data, or "content", off the web and onto our computers. We're all used to seeing this content in our browers (Chrome, Firefox, etc.). So we know that it must exist somewhere. However, it's important to realise that there are actually two ways that web content gets rendered in a browser:

1. Server-side
2. Client side

You can read here for more details (including example scripts), but for our purposes the essential features are as follows:

1. Server-side

- The scripts that "build" the website are not run on our computer, but rather on a host server that sends down all of the HTML code.
  - E.g. Wikipedia tables are already populated with all of the information — numbers, dates, etc. — that we see in our browser.
- In other words, the information that we see in our browser has already been processed by the host server.
- You can think of this information being embeded directly in the webpage's HTML.
- **Webscraping challenges:** Finding the correct CSS (or Xpath) "selectors". Iterating through dynamic webpages (e.g. "Next page" and "Show More" tabs).
- **Key concepts:** CSS, Xpath, HTML

2. Client-side

- The website contains an empty template of HTML and CSS.
  - E.g. It might contain a "skeleton" table without any values.
- However, when we actually visit the page URL, our browser sends a *request* to the host server.
- If everything is okay (e.g. our request is valid), then the server sends a *response* script, which our browser executes and uses to populate the HTML template with the specific information that we want.
- **Webscraping challenges:** Finding the "API endpoints" can be tricky, since these are sometimes hidden from view.
- **Key concepts:** APIs, API endpoints

Over the next two lectures, we'll go over the main differences between the two approaches and cover the implications for any webscraping activity. I want to forewarn you that webscraping typically involves a fair bit of detective work. You will often have to adjust your steps according to the type of data you want, and the steps that worked on one website may not work on another. (Or even work on the same website a few months later). All this is to say that *webscraping involves as much art as it does science*.

The good news is that both server-side and client-side websites allow for webscraping.[1] If you can see it in your browser, you can scrape it.

Caveat: Ethical ~~and legal~~ considerations

The previous sentence elides some important ethical considerations. Just because you *can* scrape it, doesn't mean you *should*. Now, I first have to tell you that this paragraph used to contain a warning about the legal restrictions pertaining to webscraping activity. I've decided to drop those in the wake of the landmark *hiQ Labs vs LinkedIn* court ruling. (Short version: It is currently legal to scrape data from the web using automated tools, as long as the data are publicly available.) However, it's still important to realise that the tools we'll be using over these next two lectures are very powerful. A computer can process commands much, much faster than we can ever type them up manually. It's pretty easy to write up a function or program that can overwhelm a host server or application through the sheer weight of requests.Or, just as likely, the host server has built-in safeguards that will block you in case of a suspected malicious attack. We'll return to the "be nice" mantra at the end of this lecture, as well as in the next lecture.

[1] As we'll see during the next lecture, scraping a website or application that is built on a client-side (i.e. API) framework is often easier; particularly when it comes to downloading information *en masse*.

Webscraping with rvest (server-side)

The primary R package that we'll be using today is **rvest** (link), a simple webscraping library inspired by Python's **Beautiful Soup** (link), but with extra tidyverse functionality. **rvest** is designed to work with webpages that are built server-side and thus requires knowledge of the relevant CSS selectors... Which means that now is probably a good time for us to cover what these are.

Student presentation: CSS and SelectorGadget

Time for a student presentation on CSS (i.e Cascading Style Sheets) and SelectorGadget. Click on the links if you are reading this after the fact. In short, CSS is a language for specifying the appearance of HTML documents (including web pages). It does this by providing web browsers a set of display rules, which are formed by:

1. *Properties.* CSS properties are the "how" of the display rules. These are things like which font family, styles and colours to use, page width, etc.
2. *Selectors.* CSS selectors are the "what" of the display rules. They identify which rules should be applied to which elements. E.g. Text elements that are selected as ".h1" (i.e. top line headers) are usually larger and displayed more prominently than text elements selected as ".h2" (i.e. sub-headers).

The key point is that if you can identify the CSS selector(s) of the content you want, then you can isolate it from the rest of the webpage content that you don't want. This where SelectorGadget comes in. We'll work through an extended example (with a twist!) below, but I highly recommend looking over this quick vignette before proceding.

Application 1: Wikipedia

Okay, let's get to an application. Say that we want to scrape the Wikipedia page on the **Men's 100 metres world record progression**.

First, open up this page in your browser. Take a look at its structure: What type of objects does it contain? How many tables does it have? Do these tables all share the same columns? What row- and columns-spans? Etc.

Once you've familiarised yourself with the structure, read the whole page into R using the `rvest::read_html()` function.

```
# library(rvest) ## Already loaded

m100 = read_html("http://en.wikipedia.org/wiki/Men%27s_100_metres_world_record_progression")
m100
```

```
## {html_document}
## <html class="client-nojs" lang="en" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt ns-0 ns-subject  ...
```

As you can see, this is an XML document[2] that contains *everything* needed to render the Wikipedia page. It's kind of like viewing someone's entire LaTeX document (preamble, syntax, etc.) when all we want are the data from some tables in their paper.

Table 1: Pre-IAAF (1881–1912)

Let's start by scraping the first table on the page, which documents the unofficial progression before the IAAF. The first thing we need to do is identify the table's unique CSS selector. Here's a GIF of me using SelectorGadget to do that.

```
## Sorry, this GIF is only available in the the HTML version of the notes.
```

As you can see, working through this iterative process yields *"div+ .wikitable :nth-child(1)"*. We can now use this unique CSS selector to isolate the pre-IAAF table content from the rest of the HTML document. The core **rvest** function that we'll use to extract the table content is `html_element()`, before piping it on to `html_table()` to parse the HTML table into an R data frame.

[2] XML stands for Extensible Markup Language and is one of the primary languages used for encoding and formatting web pages.

```r
pre_iaaf =
  m100 %>%
  html_element("div+ .wikitable :nth-child(1)") %>% ## select table element
  html_table()                                      ## convert to data frame

pre_iaaf
```

```
## # A tibble: 21 x 5
##     Time Athlete             Nationality    `Location of races`   Date
##    <dbl> <chr>               <chr>          <chr>                 <chr>
##  1  10.8 Luther Cary         United States  Paris, France         July 4, 1891
##  2  10.8 Cecil Lee           United Kingdom Brussels, Belgium     September 25, ~
##  3  10.8 Étienne De Ré       Belgium        Brussels, Belgium     August 4, 1893
##  4  10.8 L. Atcherley        United Kingdom Frankfurt/Main, Germ~ April 13, 1895
##  5  10.8 Harry Beaton        United Kingdom Rotterdam, Netherlan~ August 28, 1895
##  6  10.8 Harald Anderson-A~ Sweden          Helsingborg, Sweden   August 9, 1896
##  7  10.8 Isaac Westergren    Sweden         Gävle, Sweden         September 11, ~
##  8  10.8 Isaac Westergren    Sweden         Gävle, Sweden         September 10, ~
##  9  10.8 Frank Jarvis        United States  Paris, France         July 14, 1900
## 10  10.8 Walter Tewksbury    United States  Paris, France         July 14, 1900
## # ... with 11 more rows
```

Great, it worked!

I'll tidy things up a bit so that the data frame is easier to work with in R. First, I'll use the `janitor::clean_names()` convenience function to remove spaces and capital letters from the column names. (Q: How else could we have done this?) Second, I'll use the `lubridate::mdy()` function to convert the date string to a format that R actually understands.

```r
# library(janitor) ## Already loaded
# library(lubridate) ## Already loaded

pre_iaaf =
  pre_iaaf %>%
  clean_names() %>%         ## fix the column names
  mutate(date = mdy(date))  ## convert string to date format

pre_iaaf
```

```
## # A tibble: 21 x 5
##     time athlete                nationality    location_of_races       date
##    <dbl> <chr>                  <chr>          <chr>                   <date>
##  1  10.8 Luther Cary            United States  Paris, France           1891-07-04
##  2  10.8 Cecil Lee              United Kingdom Brussels, Belgium       1892-09-25
##  3  10.8 Étienne De Ré          Belgium        Brussels, Belgium       1893-08-04
##  4  10.8 L. Atcherley           United Kingdom Frankfurt/Main, Germany 1895-04-13
##  5  10.8 Harry Beaton           United Kingdom Rotterdam, Netherlands  1895-08-28
##  6  10.8 Harald Anderson-Arbin Sweden          Helsingborg, Sweden     1896-08-09
##  7  10.8 Isaac Westergren       Sweden         Gävle, Sweden           1898-09-11
##  8  10.8 Isaac Westergren       Sweden         Gävle, Sweden           1899-09-10
##  9  10.8 Frank Jarvis           United States  Paris, France           1900-07-14
## 10  10.8 Walter Tewksbury       United States  Paris, France           1900-07-14
## # ... with 11 more rows
```

Now that we have our cleaned pre-IAAF data frame, we could easily plot it. I'm going to hold off doing that until we've scraped the rest of the WR data. But first, an aside on browser inspection tools.

Aside: Get CSS selectors via browser inspection tools SelectorGadget is a great tool. But it isn't available on all browsers and can involve more work than I'd like sometimes, with all that iterative clicking.[3] I therefore wanted to mention an alternative (and very precise) approach to obtaining CSS selectors: Use the "inspect web element" feature of your browser.

Here's a quick example using Google Chrome. First, I open up the inspect console (**Ctrl+Shift+I**, or right-click and choose "Inspect"). I then proceed to scroll over the source elements, until Chrome highlights the table of interest on the actual page. Once the table (or other element of interest) is highlighted, I can grab its CSS by right-clicking and selecting **Copy -> Copy selector**.

```
## Sorry, this GIF is only available in the the HTML version of the notes.
```

In general, I prefer to obtain CSS selectors using this "inspect" method with my browser. But each to their own.

Challenge

Your turn: Download the next two tables from the same WR100m page. Combine these two new tables with the one above into a single data frame and then plot the record progression. Answer below. (No peeking until you have tried first.)

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Table 2: Pre-automatic timing (1912–1976) Let's start with the second table.

```
iaaf_76 =
  m100 %>%
  html_element("#mw-content-text > div > table:nth-child(14)") %>%
  html_table()
```

As we did with the previous table, let's fix the column names and coerce the date string to a format that R understands.

```
iaaf_76 =
  iaaf_76 %>%
  clean_names() %>%
```

[3] Historically, at least, it also had a tendency to provide CSS selectors that weren't exactly what we were looking for. To be fair, this may have reflected some issues coming from the R + **rvest** as much as anything else. These minor incompatibilities have been largely eliminated with **rvest** 1.0.0… prompting a re-write of these notes!

```
  mutate(date = mdy(date))

iaaf_76
```

```
## # A tibble: 54 x 8
##     time wind   auto athlete     nationality location_of_race   date       ref
##    <dbl> <chr> <dbl> <chr>       <chr>       <chr>              <date>     <chr>
##  1  10.6 ""       NA Donald Lip~ United Sta~ Stockholm, Sweden  1912-07-06 [2]
##  2  10.6 ""       NA Jackson Sc~ United Sta~ Stockholm, Sweden  1920-09-16 [2]
##  3  10.4 ""       NA Charley Pa~ United Sta~ Redlands, USA      1921-04-23 [2]
##  4  10.4 "0.0"    NA Eddie Tolan United Sta~ Stockholm, Sweden  1929-08-08 [2]
##  5  10.4 ""       NA Eddie Tolan United Sta~ Copenhagen, Denma~ 1929-08-25 [2]
##  6  10.3 ""       NA Percy Will~ Canada      Toronto, Canada    1930-08-09 [2]
##  7  10.3 "0.4"  10.4 Eddie Tolan United Sta~ Los Angeles, USA   1932-08-01 [2]
##  8  10.3 ""       NA Ralph Metc~ United Sta~ Budapest, Hungary  1933-08-12 [2]
##  9  10.3 ""       NA Eulace Pea~ United Sta~ Oslo, Norway       1934-08-06 [2]
## 10  10.3 ""       NA Chris Berg~ Netherlands Amsterdam, Nether~ 1934-08-26 [2]
## # ... with 44 more rows
```

Table 3: Modern Era (1977 onwards) For the final table, I'll just run the code all at once. By now you should recognise all of the commands.

```
iaaf =
  m100 %>%
  html_element("#mw-content-text > div > table:nth-child(19)") %>%
  html_table() %>%
  clean_names() %>%
  mutate(date = mdy(date))

iaaf
```

```
## # A tibble: 24 x 8
##     time wind   auto athlete nationality location_of_race date
##    <dbl> <chr> <dbl> <chr>   <chr>       <chr>            <date>
##  1 10.1   1.3     NA Bob Ha~ United Sta~ Tokyo, Japan     1964-10-15
##  2 10.0   0.8     NA Jim Hi~ United Sta~ Sacramento, USA  1968-06-20
##  3 10.0   2.0     NA Charle~ United Sta~ Mexico City, Me~ 1968-10-13
##  4  9.95 0.3      NA Jim Hi~ United Sta~ Mexico City, Me~ 1968-10-14
##  5  9.93 1.4      NA Calvin~ United Sta~ Colorado Spring~ 1983-07-03
##  6  9.83 1.0      NA Ben Jo~ Canada      Rome, Italy      1987-08-30
##  7  9.93 1.0      NA Carl L~ United Sta~ Rome, Italy      1987-08-30
##  8  9.93 1.1      NA Carl L~ United Sta~ Zürich, Switzer~ 1988-08-17
##  9  9.79 1.1      NA Ben Jo~ Canada      Seoul, South Ko~ 1988-09-24
## 10  9.92 1.1      NA Carl L~ United Sta~ Seoul, South Ko~ 1988-09-24
## # ... with 14 more rows, and 1 more variable: notes_note_2 <chr>
```

Combined eras plot

Let's combine our three separate tables into a single data frame. I'll use base R's `rbind()` to bind by row and include only the variables that are common to all of the three data frames. For good measure, I'll also add an extra column describing which era each record was recorded under.

```
wr100 =
  rbind(
    pre_iaaf %>% select(time, athlete, nationality, date) %>% mutate(era = "Pre-IAAF"),
```

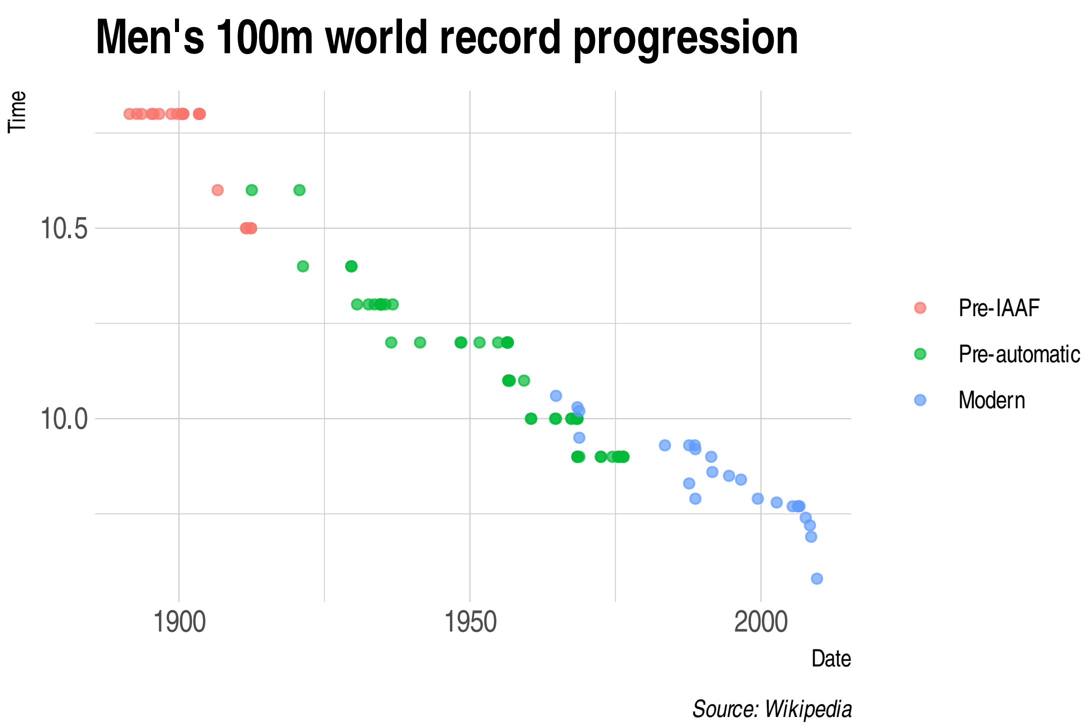
```
    iaaf_76 %>% select(time, athlete, nationality, date) %>% mutate(era = "Pre-automatic"),
    iaaf %>% select(time, athlete, nationality, date) %>% mutate(era = "Modern")
    )

wr100
```

```
## # A tibble: 99 x 5
##     time athlete                nationality    date       era
##    <dbl> <chr>                  <chr>          <date>     <chr>
##  1  10.8 Luther Cary            United States  1891-07-04 Pre-IAAF
##  2  10.8 Cecil Lee              United Kingdom 1892-09-25 Pre-IAAF
##  3  10.8 Étienne De Ré          Belgium        1893-08-04 Pre-IAAF
##  4  10.8 L. Atcherley           United Kingdom 1895-04-13 Pre-IAAF
##  5  10.8 Harry Beaton           United Kingdom 1895-08-28 Pre-IAAF
##  6  10.8 Harald Anderson-Arbin  Sweden         1896-08-09 Pre-IAAF
##  7  10.8 Isaac Westergren       Sweden         1898-09-11 Pre-IAAF
##  8  10.8 Isaac Westergren       Sweden         1899-09-10 Pre-IAAF
##  9  10.8 Frank Jarvis           United States  1900-07-14 Pre-IAAF
## 10  10.8 Walter Tewksbury       United States  1900-07-14 Pre-IAAF
## # … with 89 more rows
```

All that hard works deserves a nice plot, don't you think?

```
wr100 %>%
  ggplot(aes(x=date, y=time, col=fct_reorder2(era, date, time))) +
  geom_point(alpha = 0.7) +
  labs(
    title = "Men's 100m world record progression",
    x = "Date", y = "Time",
    caption = "Source: Wikipedia"
    ) +
  theme(legend.title = element_blank()) ## Switch off legend title
```



Application 2: Craiglist

There are several features of the previous Wikipedia example that make it a good introductory application. Most notably, the HTML table format provides a regular structure that is easily coercible into a data frame (via `html_table()`). Oftentimes, however, the information that we want to scrape off the web doesn't have this nice regular structure. For this next example, then, I'm going to walk you through a slightly more messy application: Scraping items from Craiglist.

The specific items that I'm going to scrape here are audio speakers for sale in my local city of Eugene. But you can adjust the relevant URL search parameters to your own preferences — cars in Chicago, concert tickets in Cleveland, etc. — and the same principles should carry through.

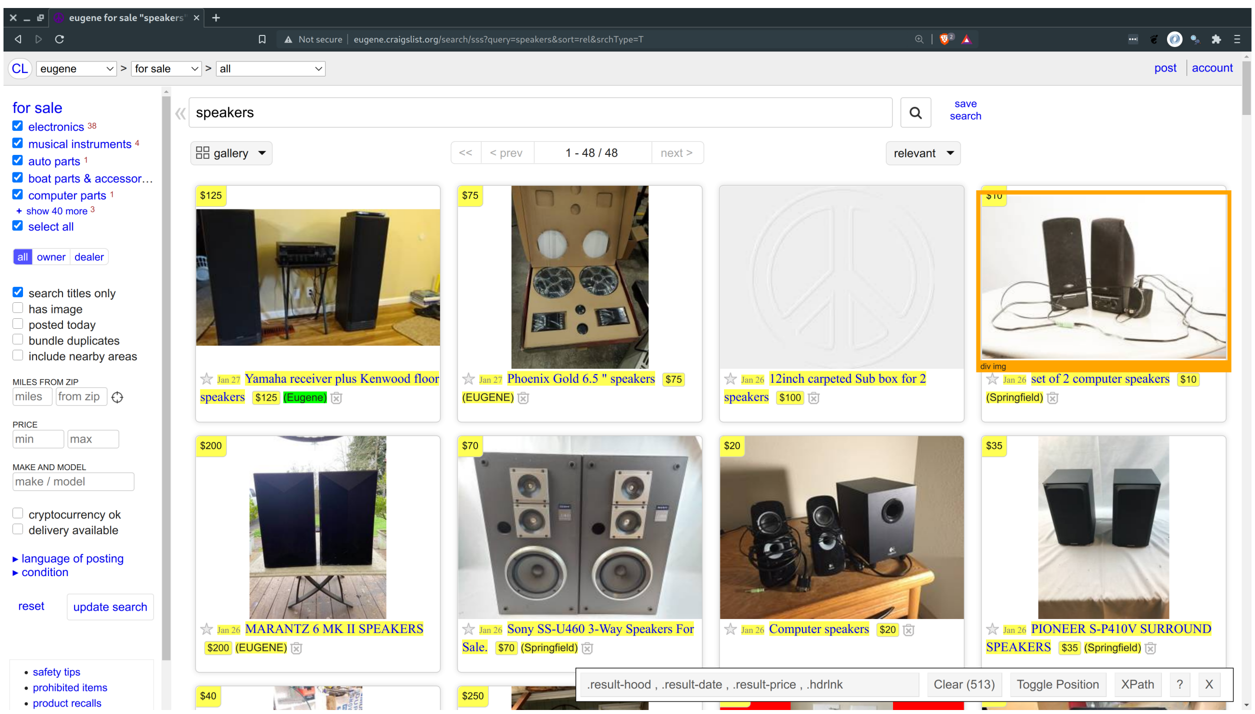
Extract the text

We start as we always do by reading in the HTML.

```
base_url = "https://eugene.craigslist.org/search/sss?query=speakers&sort=rel&srchType=T"

craiglist = read_html(base_url)
```

Next, we need to identify the CSS selectors in order to extract the relevant information from this page. Once again, this involves quite a lot of iterative clicking with SelectorGadget. I'll spare you (and myself) another GIF. But here is a screenshot of the final result once I've isolated the elements of interest. As you can see, the relevant selector is *".result-hood , .result-date , .result-price , .hdrlnk"*.



Now comes the first tweak relative to our previous example. Instead of using `html_element()`, we'll use `html_elements()` (i.e. plural) to extract *all* of the matching elements.[4] I'll assign the resulting object as `speakers`, although I won't try to coerce it to an R array just yet.

```
speakers =
  craiglist %>%
  html_elements(".result-hood , .result-date , .result-price , .hdrlnk")
```

[4]Using the singular version would simply return the very first element, which isn't very useful. Truth be told, the plural version `html_elements()` is probably a good default since it will still work with singular objects. So now you know.

At this point, you may be tempted to pipe the speakers object to html_table() to create a data frame like we did with our Wikipedia example. Unfortunately, that won't work here because we are dealing with free-form text rather than regular table structure.

```
html_table(speakers)
```

```
## Error in matrix(unlist(values), ncol = width, byrow = TRUE): 'data' must be of a vector type, was 'NULL'
```

Instead, we'll parse it as simple text via html_text(). This will yield a vector of strings, which I'll re-assign the same speakers object.

```
speakers = html_text(speakers)  ## parse as text
head(speakers, 20)              ## show the first 20 entries
```

```
##  [1] "$100"
##  [2] "Feb 10"
##  [3] "Kenwood Surround Receiver With Speakers"
##  [4] "$100"
##  [5] " (Springfield)"
##  [6] "$80"
##  [7] "Feb 10"
##  [8] "Yamaha Subwoofer With Speakers"
##  [9] "$80"
## [10] " (Eugene)"
## [11] "$100"
## [12] "Feb 10"
## [13] "Sony Home Theatre Surround Sound System: Receiver, 5 Speakers & Subwoofer - Silv"
## [14] "$100"
## [15] " (Eugene)"
## [16] "$60"
## [17] "Feb  9"
## [18] "Sony surround sound speakers and subwoofer SA-WMS230"
## [19] "$60"
## [20] " (Springfield)"
```

Coercing to a data frame

We now have a bit of work on our hands to convert this vector of strings into a usable data frame. (Remember: Webscraping is as much art as it is science.) The general approach that we want to adopt is to look for some kind of "quasi-regular" structure that we can exploit.

For example, we can see from my screenshot above that each sale item tends to have five separate text fields. (Counter-clockwise from the top: *price*, *listing date*, *description*, *price* (again), and *location*.) Based on this, we might try to transform the vector into a (transposed) matrix with five columns and from there into a data frame.

```
head(as.data.frame(t(matrix(speakers, nrow=5))))
```

```
## Warning in matrix(speakers, nrow = 5): data length [527] is not a sub-multiple
## or multiple of the number of rows [5]
```

```
##     V1     V2
## 1 $100 Feb 10
## 2  $80 Feb 10
## 3 $100 Feb 10
## 4  $60 Feb  9
## 5  $35 Feb  9
## 6   $0 Feb  9
##                                                                      V3
```

```
## 1                                          Kenwood Surround Receiver With Speakers
## 2                                                   Yamaha Subwoofer With Speakers
## 3 Sony Home Theatre Surround Sound System: Receiver, 5 Speakers & Subwoofer - Silv
## 4                            Sony surround sound speakers and subwoofer SA-WMS230
## 5                                                PIONEER S-P410V SURROUND SPEAKERS
## 6                                                      Bookshelf Speakers for sale
##      V4            V5
## 1 $100  (Springfield)
## 2  $80       (Eugene)
## 3 $100       (Eugene)
## 4  $60  (Springfield)
## 5  $35  (Springfield)
## 6   $0       (Eugene)
```

Uh-oh. This approach isn't going to work because not every sale item lists all five text fields. Quite a few are missing the location field, for instance.

Let's try a different tack. The key point about the "quasi-regular" structure that we're trying to exploit is that it needs to be present for *every* sale item. Looking again at the webpage/screenshot, can you think of something that meets that criteria?

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

How about the *listing date* text field? This is automatically populated by Craiglist and doesn't rely on the vagaries of individual seller text. In the code chunk that follows, I'm going to exploit the fact that each item includes a listing date and use that as an anchor for creating individual row entries. Now, I have to admit that I'm going to use some tools that we haven't covered yet — iteration and functions. We'll get to these important topics later on in the course, so I don't want you to stress about understanding the details of my code. The more important thing is the intuition, where I'm (a) identifying the listing date entries in the `speakers` vector, and (b) using these date entries as anchors to indicate breaks points between separate sale items. For good measure, I'm going to do this with **data.table** rather than the **tidyverse**. But that's mostly a matter of personal taste; it would be easy to translate my code to the latter if that's what you'd prefer.

```
# library(data.table) ## Already loaded

dates = as.Date(speakers, format = '%b %d')  ## Try to coerce to date of form "Jan 01"
idates = which(!is.na(dates))                ## Get index of all the valid dates (i.e. non-NA)
```

```r
## Iterate over our date index vector and then combine into a data.table. We'll
## use the listing date to define the start of each new entry. Note, however,
## that it usually comes second among the five possible text fields. (There is
## normally a duplicate price field first.) So we have to adjust the way we
## define the end of that entry; basically it's the next index position in the
## sequence minus two.
speakers_dt =
  rbindlist(lapply(
    seq_along(idates),
    function(i) {
      start = idates[i]
      end = ifelse(i!=length(idates), idates[i+1]-2, tail(idates, 1))
      data.table(t(speakers[start:end]))
    }
    ), fill = TRUE) ## Use fill=TRUE arg so that rbindlist allocates 5 cols to each row

speakers_dt
```

```
##          V1
##   1: Feb 10
##   2: Feb 10
##   3: Feb 10
##   4: Feb  9
##   5: Feb  9
##  ---
## 116: Feb 10
## 117: Feb 10
## 118: Feb 10
## 119: Feb 10
## 120: Feb 10
##                                                                           V2
##   1:                                    Kenwood Surround Receiver With Speakers
##   2:                                             Yamaha Subwoofer With Speakers
##   3: Sony Home Theatre Surround Sound System: Receiver, 5 Speakers & Subwoofer - Silv
##   4:                              Sony surround sound speakers and subwoofer SA-WMS230
##   5:                                          PIONEER S-P410V SURROUND SPEAKERS
##  ---
## 116:                  Polk Audio Vintage SDA 2B Stereo Dimensional Array Speakers Set
## 117:                                  Vintage Empire Marble Top Speakers/End Tables
## 118:                                                              Bose Speakers
## 119:                                     Pair of PA Speakers 10 inch Good Condition
## 120:                                                                       <NA>
##        V3            V4
##   1: $100  (Springfield)
##   2:  $80       (Eugene)
##   3: $100       (Eugene)
##   4:  $60  (Springfield)
##   5:  $35  (Springfield)
##  ---
## 116: $475           <NA>
## 117: $500           <NA>
## 118: $500           <NA>
## 119: $160           <NA>
```

```
## 120: <NA>           <NA>
```

Looks like it worked. Sweet.

This last bit of code is optional — and, again, I'm not going to explain myself much — but is just to tidy up the data table somewhat.

```
names(speakers_dt) = c('date', 'description', 'price', 'location')

speakers_dt[, ':=' (date = as.Date(date, format = '%b %d'),
                    price = as.numeric(gsub('\\$|\\,', '', price)))]

## Because we only get the month and day, some entries from late last year may
## have inadvertently been coerced to a future date. Fix those cases.
speakers_dt[date>Sys.Date(), date := date - years(1)]

## Drop missing entries
speakers_dt = speakers_dt[!is.na(price)]

speakers_dt
```

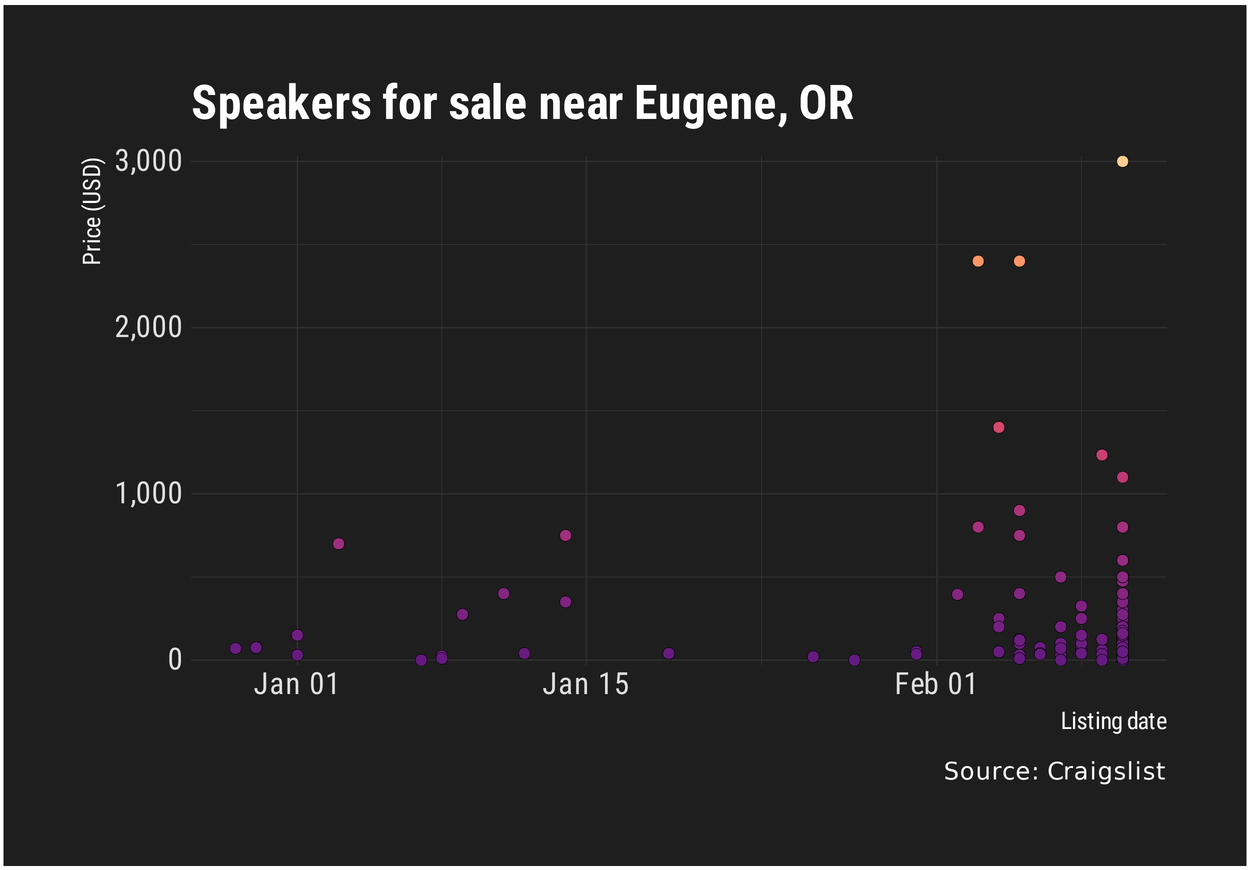
```
##            date
##   1: 2021-02-10
##   2: 2021-02-10
##   3: 2021-02-10
##   4: 2021-02-09
##   5: 2021-02-09
##  ---
## 115: 2021-02-10
## 116: 2021-02-10
## 117: 2021-02-10
## 118: 2021-02-10
## 119: 2021-02-10
##                                                                   description
##   1:                                    Kenwood Surround Receiver With Speakers
##   2:                                             Yamaha Subwoofer With Speakers
##   3: Sony Home Theatre Surround Sound System: Receiver, 5 Speakers & Subwoofer - Silv
##   4:                            Sony surround sound speakers and subwoofer SA-WMS230
##   5:                                          PIONEER S-P410V SURROUND SPEAKERS
##  ---
## 115:           (New) 2 Pairs  Klipsch Reference RCR-3  6.5" 2 Way In-Ceiling Speakers
## 116:                    Polk Audio Vintage SDA 2B Stereo Dimensional Array Speakers Set
## 117:                                Vintage Empire Marble Top Speakers/End Tables
## 118:                                                                 Bose Speakers
## 119:                                    Pair of PA Speakers 10 inch Good Condition
##      price      location
##   1:   100  (Springfield)
##   2:    80       (Eugene)
##   3:   100       (Eugene)
##   4:    60  (Springfield)
##   5:    35  (Springfield)
##  ---
## 115:   400          <NA>
## 116:   475          <NA>
## 117:   500          <NA>
## 118:   500          <NA>
```

```
## 119:   160           <NA>
```

Plot

As ever, let's reward our efforts with a nice plot. I'll add a few bells and whistles to this one, but this is most certainly optional.

```
ggplot(speakers_dt, aes(date, price)) +
  geom_point(aes(fill = price), show.legend = FALSE,
             shape = 21, colour = 'black', size = 2, stroke = 0.1) +
  scale_y_comma() +
  scale_fill_viridis_c(option = 'magma', begin = 0.3, end = 0.9) +
  labs(title = 'Speakers for sale near Eugene, OR',
       caption = 'Source: Craigslist',
       x = 'Listing date', y = 'Price (USD)') +
  theme_modern_rc()
```



Summary

- Web content can be rendered either 1) server-side or 2) client-side.
- To scrape web content that is rendered server-side, we need to know the relevant CSS selectors.
- We can find these CSS selectors using SelectorGadget or, more precisely, by inspecting the element in our browser.
- We use the `rvest` package to read into the HTML document into R and then parse the relevant nodes.
  - A typical workflow is: `read_html(URL) %>% html_elements(CSS_SELECTORS) %>% html_table()`.
  - You might need other functions depending on the content type (e.g. `html_text`).
- Just because you *can* scrape something doesn't mean you *should* (i.e. ethical and possibly legal considerations).
- Webscraping involves as much art as it does science. Be prepared to do a lot of experimenting and data cleaning.

- **Next lecture:** Webscraping: (2) Client-side and APIs.

Further resources and exercises

In the next lecture, we're going to focus on client-side web content and interacting with APIs. For the moment, you can practice your `rvest`-based scraping skills by following along with any of the many (many) tutorials available online. Lastly, we spoke a bit about the "be nice" scraping motto at the beginning of the lecture. I also wanted to point you to the **polite** package (link). It provides some helpful tools to maintain web etiquette, such as checking for permission and not hammering the host website with requests. As a bonus, it plays very nicely with the **rvest** workflow that we covered today, so please take a look.