Importing / Exporting / Webscraping

Week 3 Wednesday

Miles Chen

Department of Statistics and Data Science

UCLA

Section 1

Importing / Exporting Data

Input from a file

Basic commands

```
readline()  # for getting input from the user via stdin (the terminal)
read.table()
read.csv()
```

If you have text data, remember to use `stringsAsFactors = FALSE`

Great Import Packages in the tidyverse

```
# install.packages("readr")
# install.packages("readxl")
# install.packages("haven")
# install.packages("data.table")
library(readr)  # general file reader (for csv, txt, etc.)
library(readxl) # for importing excel files
library(haven)  # for importing SAS, SPSS, STATA
library(data.table)  # imports large tables quickly
```

Learn more about these packages at

- https://readr.tidyverse.org/
- https://readxl.tidyverse.org/
- https://haven.tidyverse.org/
- https://rdatatable.gitlab.io/data.table/

Package `readr`

`readr` supports seven file formats, each with its own read_ function:

- `read_csv()`: comma separated (CSV) files
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed width files
- `read_table()`: tabular files where columns are separated by white-space.
- `read_log()`: web log files

Package `data.table`

The package `data.table` has a function `fread()` which acts very much like `read_csv()`

It is designed for fastest performance. If you have a very massive data file (like $>$ 1GB), I recommend using `fread()`

The best source of information on how to use `data.table` is the official documentation:

- https://rdatatable.gitlab.io/data.table/

readr::read_csv() vs data.table::fread()

(Taken directly from https://readr.tidyverse.org/)

`data.table` has a function similar to `read_csv()` called `fread`. Compared to `fread`, `readr` functions:

- Are slower (currently ~1.2-2x slower). If you want absolutely the best performance, use data.table::fread().
- Use a slightly more sophisticated parser, recognising both doubled (“ “ “ “) and backslash escapes (” ”“), and can produce factors and date/times directly.
- Forces you to supply all parameters, where `fread()` saves you work by automatically guessing the delimiter, whether or not the file has a header, and how many lines to skip.
- Are built on a different underlying infrastructure. Readr functions are designed to be quite general, which makes it easier to add support for new rectangular data formats. `fread()` is designed to be as fast as possible

Package readxl

```
library(readxl)
read_excel("datasets.xls") # will read in the first worksheet
# you can specify a different worksheet by name or number
read_excel("datasets.xls", sheet = "mtcars")
read_excel("datasets.xls", sheet = 2)
```

Downloading files

Occasionally, you'll want to download a data file from the internet. R's native `download.file()` function can achieve this.

```
url <- "https://raw.githubusercontent.com/smileschen/playground/master/iris.csv"
download.file(url, destfile = "iris.csv")  # files will get saved to your working dir
iris <- readr::read_csv("iris.csv")
```

Saving and Exporting Data

You can save objects to files for reuse.

```
save(object1, object2, ... , file = "object.RData") # native .RData format
write(x, "file.txt", ncol = 1) # saves atomic vector as plain text
write.csv(df, file = "df.csv") # saves a data.frame to csv file
write.csv(df, file = "df.csv", row.names = FALSE) # removes row names
```

Package lubridate

Handling Date and Time info in R can be tedious, frustrating, and painful

The `lubridate` package make getting date info into R much easier.

Recommended Reading: https://lubridate.tidyverse.org/

Lubridate Cheat Sheet: https://rawgit.com/rstudio/cheatsheets/main/lubridate.pdf

Without Lubridate

```
sdate1 <- "January 15, 1999"
as.Date(sdate1, "%B %d, %Y") # formatting match perfectly or it will fail
```

```
[1] "1999-01-15"
```

```
as.Date(sdate1, "%B %d %Y") # comma missing leads to NA
```

```
[1] NA
```

```
sdate2 <- "12-15-2001"
as.Date(sdate2, "%m-%d-%Y")
```

```
[1] "2001-12-15"
```

```
as.Date(sdate2, "%m %d %Y") # no dashes leads to NA
```

```
[1] NA
```

Without Lubridate

```
sdates <- c("January 15, 1999", "12-15-2001", "03/18/2002")
as.Date(sdates,"%B %d, %Y") # only one format at a time
```

```
[1] "1999-01-15" NA           NA
```

```
as.Date(sdates,"%m-%d-%Y")
```

```
[1] NA           "2001-12-15" NA
```

```
as.Date(sdates,"%m/%d/%Y")
```

```
[1] NA           NA           "2002-03-18"
```

Lubridate

```
# install.packages("lubridate")
library(lubridate)
sdates <- c("January 15, 1999", "12-15-2001", "03/18/2002")
mdy(sdates) # Can parse a vector of dates with for different formats
```

```
[1] "1999-01-15" "2001-12-15" "2002-03-18"
```

Does require that all dates are written in same mdy order

Lubridate

Lubridate requires you to specify the order of the fields.

```
sdate3 <- "03/04/05" # ambiguous date
ymd(sdate3)
```

```
[1] "2003-04-05"
```

```
mdy(sdate3)
```

```
[1] "2005-03-04"
```

```
dmy(sdate3)
```

```
[1] "2005-04-03"
```

```
mdy("25-12-99") # Will return NA if it can't parse it
```

```
[1] NA
```

Package rvest

```
# install.packages("rvest")
library(rvest)
```

"harvesting" from the web, aka. web scraping

Documentation: https://rvest.tidyverse.org/

Recommended Reading: https://rvest.tidyverse.org/articles/rvest.html

Use Selector Gadget with rvest:

https://rvest.tidyverse.org/articles/selectorgadget.html

rvest

rvest is great for fairly static webpages with well defined html tags.

rvest does not work as well for sites that are generated via javascript, e.g. sites with infinite scroll.

rvest does not work well for sites where you must be logged in to view content.

HTML and CSS

A very brief understanding of HTML can help you understand how rvest works.

Webpages are written in a markup language called HTML. Formatting is applied by using tags. To keep a page's format consistent, the designer will often use CSS classes and in a separate file specify how those classes should be displayed.

HTML and CSS

Here's a simplified version of what those tags may look like in the underlying HTML.

```
<ul class="flat-list buttons">
    <li class="first">
        <a href="#" class="comments">4612 comments</a>
    </li>
    <li class="share-button">
        <a href="#">share</a>
    </li>
    <li class="save-button">
        <a href="#">save</a>
    </li>
    <li class="hide-button">
        <a href="#">hide</a>
    </li>
    <li class="report-button">
        <a href="#">report</a>
    </li>
</ul>
```

Selector Gadget

The selector gadget works by looking at the CSS tags and selecting all tags on the webpage that has a specific tag.

I go to the site and click text that I want to capture. The selector gadget highlights everything it will capture in yellow. If there is stuff I do not want, I click that and selector gadget will adjust the tag to unselect.

So if I use the tag `.comments`, the selector gadget will identify all text that is surrounded by a tag with the class `.comments`. [Don't worry about the fact that there's a dot, the selector gadget will determine if there should or should not be a dot included in the tag to use.]

rvest

With the tag, we can go back to rvest and select the nodes.

```
old_reddit <- read_html("https://old.reddit.com/")
comment_counts <- old_reddit |>
  html_nodes(".comments") |>
  html_text()
comment_counts
```

```
 [1] "424 comments"  "4433 comments" "6536 comments" "5371 comments"
 [5] "500 comments"  "4373 comments" "2831 comments" "2732 comments"
 [9] "2621 comments" "398 comments"  "212 comments"  "2306 comments"
[13] "2288 comments" "2222 comments" "2180 comments" "1916 comments"
[17] "2172 comments" "332 comments"  "86 comments"   "314 comments"
[21] "2073 comments" "1991 comments" "1994 comments" "342 comments"
[25] "1919 comments"
```

Using your scraped text

We can now extract these values with regular expressions (covered later).

```
library(stringr)
as.numeric(str_extract(comment_counts, "\\d+"))
```

```
 [1]  424 4433 6536 5371  500 4373 2831 2732 2621  398  212 2306 2288
[14] 2222 2180 1916 2172  332   86  314 2073 1991 1994  342 1919
```

Hard for rvest

Sites featuring infinite scroll don't work well with rvest. For example, reddit.

Sites that require logging in to view content also do not work well. For example, Facebook, X/twitter.

Easier for rvest

rvest works best with fairly static, public pages.

The css selector

```
old_reddit <- read_html("https://old.reddit.com/")
old_reddit |>
  html_nodes(".title.may-blank") |>
  html_text() |>
  head(10)
```

```
 [1] "Trump Added $2.25 Trillion to the National Debt in His First Year Back
 [2] "favorite ugly actors?"
 [3] "Greenlanders are trolling the US by pretending to be fentanyl addicts"
 [4] "Greenland Leader Tells People to Prepare for Possible Invasion"
 [5] "Lived here two years. Got our water heater replaced today, and suddenly
 [6] "Greenlanders are trolling the US by pretending to be fentanyl addicts"
 [7] "Air cooling is better than Liquid cooling"
 [8] "AITAH for ruining a wedding?"
```

Section 2

Polite scraping with {polite}

Package polite

Sometimes the HTML is accessible, but you still want to:

- **identify yourself** with a clear user-agent
- **respect robots.txt**
- **rate-limit requests** so you don't hammer a server
- **reuse a session** (cookies/headers), instead of "cold" requests each time

That's what **{polite}** is for: it wraps rvest/http requests in a "be nice on the web" workflow.

What {polite} does (and does not) do

{polite} helps you be a good citizen and manage a session:

- checks `robots.txt` for your user-agent (permission)
- enforces a delay between requests (rate limiting)
- lets you "move around" a site with `nod()`

polite does not run JavaScript. If the content truly only appears after JS executes, you'll need a browser automation tool (e.g., chromote/RSelenium), not just polite+rvest.

Using polite: bow, nod, scrape

A polite session has three steps:

1. **bow(url, ...)** Introduce yourself and check `robots.txt`. Returns a session object.
2. **nod(session, path)** Update the session to a new path on the same host.
3. **scrape(session, ...)** Fetch the content (HTML/JSON/etc.), returning something you can parse (often an HTML document for rvest).

delay, times, and force

- `delay` = desired seconds between requests (polite may increase it if robots.txt requires more)
- `times` = how many times to retry a request
- `force = TRUE` = proceed even if robots rules say "no" (generally **avoid** unless you have permission)

Other Webscraping Options

To work with javascript generated pages, you can use `RSelenium`, which is powerful but has a steeper learning curve.

Other language options:

- Python Beautiful Soup - very easy to learn, but works only on static pages.
- Python Scrapy - more complex, can crawl across many pages, including dynamic pages. Steeper learning curve.
- Python Selenium - supports javascript generated pages. Most complex to learn.

Section 3

rvest demo