The purpose of this section is to showcase the ability to scrape and process web data using `R`.

Last section

Simulate data with a given variance-covariance matrix

Let's take another crack at this. Let Σ be our $k \times k$ given covariance matrix. We want to generate data $\mathbf{X}$ from some hypothetical population $\mathbf{P}$ s.t. $V(\mathbf{P}) = \Sigma$ and $V(\mathbf{X}) \approx \Sigma$.

Let $\mathbf{C}'\mathbf{C} = \mathbf{\Sigma}$ ($\mathbf{C}$ is the cholesky decomposition of Σ) and let $\mathbf{Z}$ be an $n \times k$ matrix where each element is a pull from a standard normal distribution. We know that $V(\mathbf{Z}) \approx \mathbf{I}_k$. Using the properties of the variance function and that Σ and $\mathbf{Q}$ are just $k \times k$ matrices of constants, we get:

$$\begin{aligned} V(\mathbf{P}) &= \Sigma \\ &= \mathbf{Q}'\mathbf{Q} \\ &= \mathbf{Q}'I_k\mathbf{Q} \\ &\approx \mathbf{Q}'V(\mathbf{Z})\mathbf{Q} \\ &= V(\mathbf{ZQ}) \equiv V(\mathbf{X}) \end{aligned}$$

Which is exactly what we did in code last week.

R^2 in IV/2SLS

You already know that R^2 values are not to be trusted. What about R^2 values in an IV/2SLS regression? Even worse! Why? We know that

$$R^2_{OLS} = 1 - \frac{\text{SSR}}{\text{SST}} = 1 - \frac{\sum_i (y_i - \mathbf{x}_i \mathbf{b}_{OLS})^2}{\sum_i (y_i - \bar{y})^2}$$

To calculate the R^2_{2SLS}, we simply substitute $\mathbf{b}_{2SLS}$ for $\mathbf{b}_{OLS}$ in the above equation. We do not, however, substitute in $\mathbf{z}_i$ for $\mathbf{x}_i$, so the numerator is not our residuals from the second-stage regression (which we know to be incorrect, as Max covered in lecture). Because we don't make that substitution, we no longer have the guarantee that SSR < SST[1].

Even worse, the R^2_{2SLS} is even less useful than the R^2_{OLS} because $V(\mathbf{y}) = V(\mathbf{X}\mathbf{b}_{2SLS} + \varepsilon) = V(\mathbf{XB}) + V(\varepsilon) + 2\text{Cov}(\mathbf{Xb}, \varepsilon)$, so our R^2_{2SLS}, which is supposed to describe $1 - \frac{V(\varepsilon)}{V(\mathbf{y})}$, no longer has any meaning.

Web scraping

The purpose of this section is to showcase the ability to scrape and process web data using `R`. The section notes draw heavily from a post on a great blog by Pascal Mickelson and Scott Chamberlain,

[1]Another way to say this is that the residuals we calculate here do not come from a model that nests the constant-only model of y.

two biologists and experienced `R` users.

Note that the code below actually takes quite a while to run, since we need to download multiple large files from the server. If you are reading this in section while I'm demonstrating it, I do not recommend you run it! We don't want to crash their server. For the purposes of section I will stop the loop below after one run, but I leave it in its full form here for demonstration purposes.
Suppose we want to find the number of available economics journals. There are too many. Definitely. But suppose we want to find out just how many. To do this, we can visit crossref.org, which is a citation-linking network with a list of all journals and their Digital Object Identifiers (DOIs). We will query the list from within `R` and then parse the returned content to list journals with certain attributes. For this, we'll need to load(/install) the following libraries:

```
library(XML)
library(RCurl)
library(stringr)

options(show.error.messages = FALSE)
```

Note the useful option for code with loops, especially loops over remote queries, to globally suppress error messages. The next step is to repeatedly query crossref.org for journal titles. Try to copy and paste the base URL address (`baseurl`) into your browser:

`http://oai.crossref.org/OAIHandler?verb=ListSets`.

The result is a long XML form. The idea behind scraping data is to take web data like this and turn it into a dataset we can analyze. The function `getURL` in the following code pulls this response into `R` as a string, and the outer functions `xmlParse` and `xmlToList` convert the output into an `R` data structure. There are too many entries to fit into a single query, so the `while` loop continues to query until there are no more results. The final results are stored in `nameslist`.

```
token <- "characters"
nameslist <- list()

while (is.character(token) == TRUE) {
  baseurl <- "http://oai.crossref.org/OAIHandler?verb=ListSets"
  if (token == "characters") {
    tok.follow <- NULL
  } else {
    tok.follow <- paste("&resumptionToken=", token, sep = "")
  }

  query <- paste(baseurl, tok.follow, sep = "")

  xml.query <- xmlParse(getURL(query))
  set.res <- xmlToList(xml.query)
  names <- as.character(sapply(set.res[["ListSets"]], function(x) x[["setName"]]))
  nameslist[[token]] <- names
  if (class(try(set.res[["request"]][[".attrs"]][["resumptionToken"]])) == "try-error") {
    stop("no more data")
  }
  else {
    token <- set.res[["request"]][[".attrs"]][["resumptionToken"]]
  }
}
```

This looks confusing. Don't panic[2], though. A `while` loop runs repeatedly as long as the condition in parenthesis is satisfied. Here's what it does:

1. The `while` loop creates a url, `query` to pull from the URL above.
2. Gets the results, `xml.query`, and runs `xmlToList()` to turn those XML results into a list of journals, `set.res`.
3. Passes the names of the journals, `names`, to the list of names, `nameslist`. The list identifier for each set of names is the `token`.
4. Tries to set the `token` variable equal to the "resumptionToken" field passed in the XML.
   (a) If successful, restarts loop at 1 using the new `token`.
   (b) If unsuccessful, ends loop.

How many journal titles are collected by this query? We first concatenate the results into a single list, and then find the total length:

```
allnames <- do.call(c, nameslist)
length(allnames)
head(allnames)
```

```
: [1] 27289
```

`do.call()` is a way to combine all of the different lists of journals within `nameslist` into one.

Now, suppose that we are looking for just those journals with *economic* in the title. We rely on regular expressions, a common way to parse strings, from within `R`. The following code snippet detects strings with some variant of *economic*, both lower- and upper-case, and selects those elements from within the `allnames` list.

```
econtitles <- as.character(allnames[str_detect(allnames, "^[Ee]conomic|\\s[Ee]conomic")])
length(econtitles)
```

```
: [1] 461
```

What in the hell? So many! I suppose that this is a good thing: at least one of the 461 journals should accept my crappy papers. If I blindly throw a dart in a bar, it may not hit the dartboard, but it will almost certainly hit one of the 461 patrons. Here is a random sample of ten journals:

```
sample(econtitles, 10)
```

[2]Don't panic is officially the unofficial ARE212 section motto, by the way.

```
 [1] "Computer Science in Economics and Management"
 [2] "Advances in Theoretical Economics"
 [3] "Economic Systems"
 [4] "Journal of Agricultural Economics"
 [5] "Economic Quality Control"
 [6] "Contributions in Economic Analysis & Policy"
 [7] "Journal of Economic Interaction and Coordination"
 [8] "Asian Economic Journal"
 [9] "International Review of Economics"
[10] "African Journal of Economic Policy"
```

What are other things we can do with the data? Suppose we wanted to compare the relative frequencies of different subjects within journal titles. This offers a decent example for section, since we can refactor some of the code we already developed — a useful skill for writing clean code. We have already figured out how to count the number of journals for a particular regular expression. We can refactor the code into the following function, which accepts a regular expression and returns the length of the collection containing matching strings:

```
countJournals <- function(regex) {
  titles <- as.character(allnames[str_detect(allnames, regex)])
  return(length(titles))
}
```

Now the tedious process of converting a list of subjects into the appropriate regular expressions[3]. If we have time, we'll write a function to do this conversion for us:

```
subj = c("economic", "business", "politic", "environment", "engineer", "history")
regx = c("^[Ee]conomic|\\s[Ee]conomic", "^[Bb]usiness|\\s[Bb]usiness",
  "^[Pp]olitic|\\s[Pp]olitic", "^[Ee]nvironment|\\s[Ee]nvironment",
  "^[Ee]ngineer|\\s[Ee]ngineer", "^[Hh]istory|\\s[Hh]istory")

subj.df <- data.frame(subject = subj, regex = regx)
```

Finally, we simply apply our refactored function to the regular expressions, and graph the result:

```
subj.df$count <- sapply(as.character(subj.df$regex), countJournals)
(g <- ggplot(data = subj.df, aes(x = subject, y = count)) + geom_bar())
```

Web scraping in the real world

Practically, every web scraping task requires a different set of tools. The example presented here is one of the most straightforward possible settings, since we're scraping an XML file. The only slight complication was using the `token` to get the next set of results.

Second, web scraping is a neat trick, but I recommend only using it as a last resort. If you can get the data you want just by asking, that's *much* easier. Moreover, some website owners may not appreciate your hammering their site with requests. Finally, web scraping is fragile and heavily dependent on the site not changing over time. This is particularly true when scraping HTML, perhaps somewhat less so with XML, since the latter tends to have a fairly well-defined structure to it. Still, nothing is set in stone.

[3]Unfortunately, we don't have time to do a detailed introduction to regular expressions, but they *are* incredibly useful. You can find a gentle introduction here: `http://www.zytrax.com/tech/web/regex.htm`.

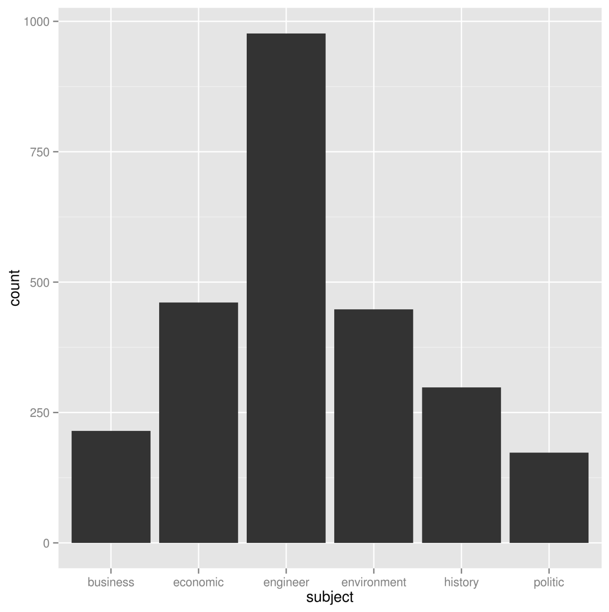

Figure 1: Journal count by subject search