Homework 9

Setting up: more packages (yak shaving)

The two most common sources for R packages are CRAN and Github. CRAN is specific to R; Github is a "social code" site of enormous size that has become the de facto standard host for open source software in current development. RStudio's "Install Packages…" menu command or the R `install.packages` function are primarily for installing from CRAN. You can simply enter `install.packages("XML")` and R will look on CRAN for the `XML` package, download it, and install it. Other packages are not in CRAN. The `devtools::install_github` is a convenient way to get a package on Github. The syntax is `install_github("user/repo")` where `user` is the name of the user and `repo` is the name of the package repository. This also corresponds to the part of the URL that follows `http://www.github.com/`. For example, the package for this course is http://www.github.com/agoldst/litdata. You have been installing it with `install_github("agoldst/litdata")`. As this example suggests, when you load the package, regardless of where you installed it from, you use only the name of the package, as in `library("litdata")`.

This is as good a moment as any to remark about the `xyz::abc` syntax. It means "`abc` from the package `xyz`", and you can use it to access a function (or other variable) in the `xyz` package even without loading that package using `library`.

Install the following additional packages from CRAN. Please let me hear from you soon if you encounter problems with the installation process:

- `XML`
- `httr`
- `jsonlite`
- `rvest`

Incidentally, CRAN also has a set of pages which suggest good packages for various kinds of work you might want to do in R: these are called the Task Views. There is one for Web Technologies.

Extra notes on working with XML in R (no exercises)

Memory leaks, segmentation faults, and other happy accidents

`XML` is one of the many packages which allow R programs to interface with software written in other programming languages. In this case the software is libxml2, a C-language library for processing XML. C is less user-friendly than R and abstracts away fewer operations of the machine, but it can be much faster. In particular, in C, the programmer must manually keep track of where all variables are stored and when they can be "cleaned up" so that the RAM used to store them can be used for something else. In R, you normally never worry about this, because the language takes care of it for you through an automatic process charmingly known as *garbage collection*.

Because your `XML` objects (which are returned from functions like `xmlParse`) are actually living over in a C program, it's possible to write an R program that either fails to clean them up when it is done with them (in which case they sit in memory, taking up space: this is a *memory leak*) or that tries to access data which has already been deleted. In the latter case, a *segmentation fault* occurs and your program will crash, taking R with it. (RStudio itself will normally not crash: it will show you a cute "exploding bomb" dialog box and then restart, but you will have lost anything you didn't save.)

Don't worry about memory leaks at this stage; if you spend a lot of time trying out `XML` functions in your console and things get really slow, you might be able to fix the situation by quitting RStudio. And if you are using R interactively, it's hard to occasion a segfault unless you use the `XML` function `free`, which deletes the C data structures associated with an R XML variable. But if you are knitting, it is *all too easy*, because `knitr` tries to cache the results of your computations, but does not keep track of R values that are secretly pointers to C values. If you knit, change your R markdown, then knit again, your program may try to access cached pointers to values that no longer exist. Long story short: if you use `XML` in your R markdown you can encounter mysterious segfault crashes when you knit. The way to avoid this is to disable cacheing for chunks that refer to `XML` objects (this includes document objects from `xmlParse`, nodes from `xmlRoot`, and node sets from `getNodeSet`). To do this, simply add `cache=F` to the chunk options. Or, to turn off cacheing globally for your whole program, add `cache=F` to the options passed to `knitr::opts_chunk$set` in your first chunk. I recommend this as the simpler option, but do this only for programs that use `XML` or other packages that interface with outside software, so that you can still benefit from the big speed-up cacheing provides when you are writing your programs. (There are fancier approaches to this issue, but you spend more time figuring them out than you lose by just disabling cacheing.)

Nodeset indexing

I wasn't clear enough in class about ways to subscript collections of XML nodes. In particular, you *can* use logical subscripting with nodesets. As with lists, there are two subscript operators: sub-list subscripting with single brackets `[...]` and single-element subscripting with double brackets `[[...]]`. You can use vector subscripts only with `[...]`, and the results are again a collection (even of a single element). With `[[...]]` you can only use a single numerical index or a name in quotes (if the nodes are named, as in the results from `xmlChildren`), and the result is a single node value. Resuming the example from class:

```r
crisis <- xmlParse("tei-sample/mjp/Crisis130_22.2.tei.xml")
# document object can be treated as though it were
# itself the root node
all_divs <- getNodeSet(crisis, "//def:div",
                       namespaces=c(def="http://www.tei-c.org/ns/1.0"))
div_types <- xmlSApply(all_divs, xmlGetAttr, "type")
```

We can use single-bracket logical-vector subscripting to pick out the `<div type="poetry">` element:

```r
all_divs[div_types == "poetry"]
```

```
[[1]]
<div type="poetry">
  <ab>THE NEGRO SPEAKS OF RIVERS </ab>
  <ab>LANGSTON HUGHES </ab>
 <ab>I'VE known rivers: I've known rivers ancient as the world and older than the flow of human blood in human
  <ab>My soul has grown deep like the rivers. </ab>
  <ab>I bathed in the Euphrates when dawns were young. </ab>
  <ab>I built my hut near the Congo and it lulled me to sleep. </ab>
  <ab>I looked upon the Nile and raised the pyramids above it. </ab>
 <ab>I heard the singing of the Mississippi when Abe Lincoln went down to New Orleans, and I've seen its muddy
  <ab>I've known rivers; Ancient, dusky rivers. </ab>
  <ab>My soul has grown deep like the rivers. </ab>
</div>
```

The result is a single-element list (an ordinary R list) holding the `<div>` node containing Hughes's poem. (The last two code blocks, incidentally, are `cache=F` chunks in my R markdown.)

XML functions: A quick reference

`xmlParse(filename)` Reads in an XML document stored in `filename`.

`xmlTreeParse(filename)` Same as `xmlParse`, but, by default, return a nested R list representing the XML hierarchy instead of using the C data structures. Slower but sometimes convenient.

`xmlRoot(doc)` Extracts the root node of the results of an `xmlParse`. Often superfluous (since in general a document is treated as synonymous with its root node by many of the other `XML` functions).

`xmlChildren(node)` Gets an ordinary R list of `node`'s child nodes.

`xmlParent(node)` Gets the unique parent node of `node`.

`xmlValue(node, recursive=T)` If `node` is a leaf (with no children), returns the text in `node`. Otherwise, if `recursive` is `TRUE` (as it is by default), tries to stick together all the text in all the leaves that are descendants of `node`.

`xmlAttrs(node)` Gets a vector containing all the attributes of `node` (the names of the vector elements are the attribute names).

`xmlGetAttr(node, attr, default)` Gets the value of the `attr` attribute of `node`. If there is no such attribute, returns `default` instead. It's sometimes convenient for this to be something other than `NULL`.

`names(node)` Returns a vector of the names of the children of `node`.

`node[[j]]` Returns the `j`th child of `node`.

`node[index_vector]` Returns the subset of `node`'s children as given by `index_vector`.

`getNodeSet(doc, xpath, ns)` Searches the *document containing* `node` for all nodes matching the XPath given in the string `xpath`. If `xpath` is relative then the search is relative to `node`. `ns` is a named character vector giving namespace shorthands used in `xpath`. The result is a list-like object called a "node set."

`xmlApply(nodeset, f, ...)` Returns the list resulting from calling `f` on each node in `nodeset` with additional parameters in `...`.

`xmlSApply(nodeset, f, ...)` Like `xmlApply`, but if each result is a single value, then the result is an ordinary vector.

`xpathApply(doc, xpath, f, ..., ns)` More efficient shortcut for `xmlApply(getNodeSet(doc, xpath, ns), f, ...))`.

`xpathSApply` Simplifying version of `xpathApply`.

As an example of `xpathApply` and some of the other operations above, I'll give a somewhat speedier revision to an example from class. Let's define a function to operate on a `<sp>` (speech) node, `process_node`, by extracting the name of the speaker and the `met` attributes of all the children `<l>` elements, and combining these into a little two-column data frame.

```
process_node <- function (sp) {
    speaker <- xmlGetAttr(sp, "who",
                          default="<missing>")
    meter <- sapply(sp[names(sp) == "l"],
                    xmlGetAttr, "met",
                    default="<missing>")
    data.frame(speaker=speaker, meter=meter,
               stringsAsFactors=F)
}
```

Now we can load the file:

```
fe <- xmlParse("fair-em/A21328-sheriko.xml")
```

In an XPath, `x[y]` means "an `x` node with at least one `y` child." So our procedure will be to select all the `<sp>` nodes with at least one `<l>` child and then apply `process_node` to them:

```
# annoying namespace gotcha
ns <- c(def="http://www.tei-c.org/ns/1.0")
meters_list <- xpathApply(fe, "//def:sp[def:l]", process_node,
                          namespaces=ns)
```

`meters_list` is a list of our data frames, one frame for each speech in the play. To combine them into a big data frame, we use:

```
spkrs_meter <- do.call(rbind, meters_list)
```

And then we can proceed to further analysis as in class.

Jockers: a little more XML practice (exercises)

Read Jockers, chap. 10. At last you can make use of some of the other supplied data in the `TextAnalysisWithR.zip` archive. Jockers continues with a TEI-encoded *Moby-Dick*, which you can find in `TextAnalysisWithR/data/XML1/melville1.xml`. In the `TextAnalysisWithR/data/XMLAuthorCorpus` folder, he has supplied a number of other interesting texts, including several Irish-American novels, e.g. *Black Soil* by Josephine Donovan (1930). But just for the purposes of the simple exercise in programming, let's work with `melville1.xml`. Copy it into your folder for this homework and read it in using

```
mb <- xmlParse("melville1.xml")
```

The chapter reviews some of the `XML` functions you've already seen. Jockers uses `xmlTreeParse` with `useInternalNodes=T`; as the `XML` help tells you, you can just as well use `xmlParse` with no extra parameters with the same result. Jockers uses the `xmlElementsByTagName` function, which you can read about in the online help, but you don't need it. Use `[...]` and `names` to save typing. Finally, in section 10.6, Jockers uses `xpathApply` rather idiosyncratically: it turns out that if you omit the function to apply to the node set, `xpathApply` works like `getNodeSet`. You can verify this by replacing all the 10.6 uses of `xpathApply` with `getNodeSet`.

Skip the 10.1 practice problem.

It won't surprise you that we can produce a rather more efficient and modular version of the big `for` loop on p. 94, in which, in fact, we don't explicitly use `for` at all. Write a function, `process_chap`, that takes a `div1` node, extracts all the text in each of its `p` children, and produces a data frame with two columns, `chapter` and `feature`. The `chapter` should come from the `n` attribute of the `div1`; the `feature` should be (yet again) a vector of all the words of the chapter. Here's your friend `featurize`, which you should use within `process_chap`:

```
featurize <- function (ll) {
    result <- unlist(strsplit(ll, "\\W+"))
    result <- result[result != ""]
    tolower(result)
}
```

Hint: use `process_node` above as a model. You can follow it quite closely, but you need to change one of the functions used; think about what parts of the XML you need to extract from which `div1`. Test if your `process_chap` function works by applying it to this miniature example:

```
mini <- xmlParse(
'
<div1 n="1"><p>Things happen.</p><p>Often <emph>more</emph> than once.</p></div1>
')
process_chap(xmlRoot(mini))
```

```
  chapter feature
1       1  things
2       1  happen
3       1   often
4       1    more
5       1    than
6       1    once
```

Now use `xpathApply` to get all `div1` nodes of type `chapter`, and process them with `process_chap`, yielding a list of data frames of features in each chapter, `chap_frames`. *Hints*: beware the namespace gotcha. Also, the XPath you need to get those `div1` nodes can be found in Jockers's code, but he uses it in a call to `getNodeSet` instead of `xpathApply` (which is why he then writes a big `for` loop).

Now create one big data frame (one row for each word in the body chapters of the novel!) with:

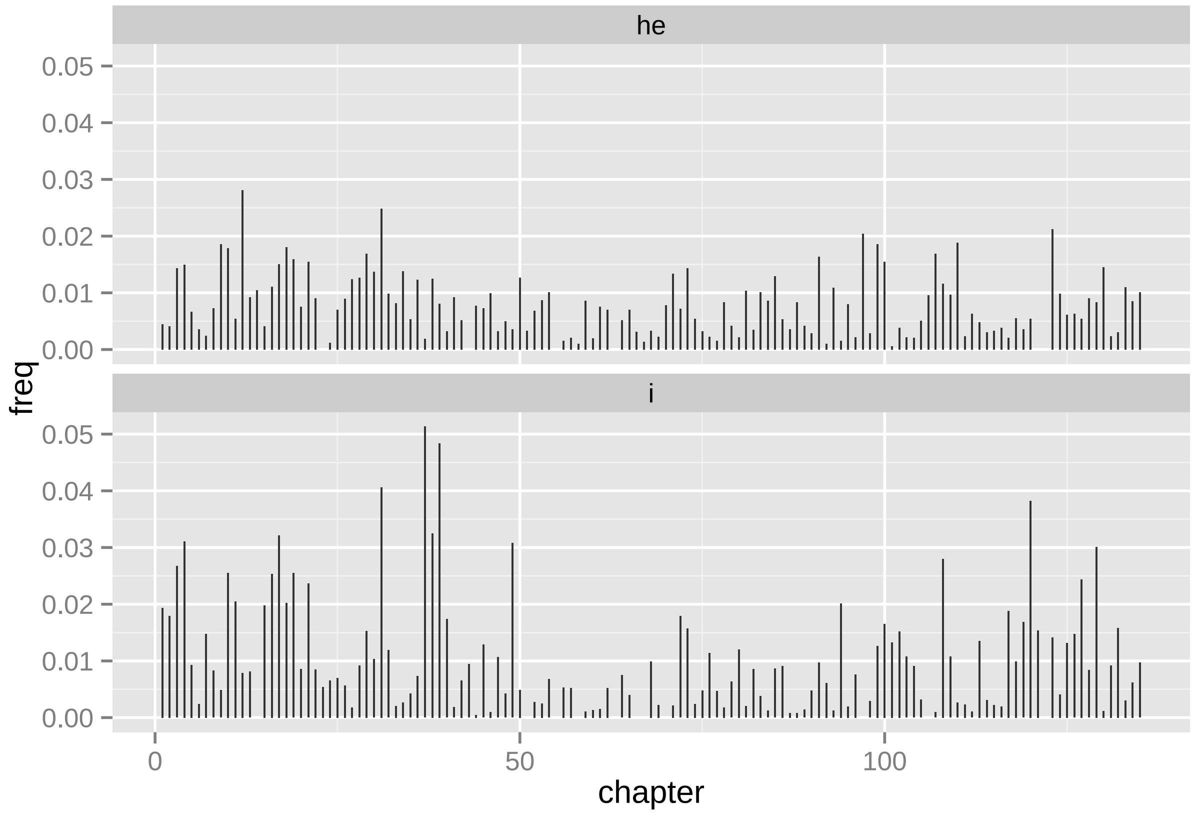
```
mb_features <- do.call(rbind, chap_frames)
```

As it happens the `chapter` column is still character data. Let's assert that it is in fact numeric:

```
mb_features$chapter <- as.numeric(mb_features$chapter)
```

Now let's make a graph on the model of Jockers's Figure 10.1. But enough of `ahab` and `whale`. Let's count the pronouns `i` and `he` instead. Use `dplyr` to create a table with per-chapter totals of each feature; then add a per-chapter *frequency* for each feature (number of occurrences divided by total of all words in the chapter); then filter to keep only your key words `i` and `he`; you should end up with two *rows* for each chapter and columns `chapter`, `feature`, and `freq`. Then create a faceted bar plot (instead of the side-by-side `barplot` Jockers uses). *Hints*: this is another bar graph where you need to set `stat="identity"`; also, you can ignore warnings about `position_stack`.

Here's mine (I put the plots on top of one another instead of side-by-side and made the bars thinner with a constant `width` value):



Very optional: use `tidyr::spread` to produce a data frame suitable to try out the `cor.test` call at the end of 10.5. `cor.test(x, y)`, by the way, gives statistical information about the fit of the linear regression of `y` on `x`.

HTML

Another markup language (no exercise)

You already know a lot about HTML, even if you have never learned anything about HTML. First of all, you interact with HTML all the time, since it is the language in which web pages are encoded. Second of all, HTML is more or less a specialized subset of XML, so you already know the basic HTML principles: an ordered hierarchy, specified by tags within `<...>` with possible attributes. The root of an HTML document is an `<html>` element; its children are a `<head>` and a `<body>`. Use your browser's "View Source" menu command to look over any webpage you like (often you have to scroll past a lot of gobbledygook in the `<head>` to find the `<body>` holding the text you read on the page) and get a sense of this structure.

And third! Markdown is in fact designed to be convertible to HTML, and so you already know quite a lot about the kinds of markup HTML uses—just via its Markdown equivalents instead of the canonical HTML tags. Markdown does italics with `*...*`; in HTML, the equivalent is `<em>...</em>`. Markdown section headings are `# ...` in HTML, the equivalent is `<h1>...</h1>`. Paragraphs are separated by blank lines in markdown; in HTML, more verbosely, you use `<p>...</p>` tags.

As for the "hyper" part of "hypertext," most of the work is done by one more tag, `<a>`. An `<a>` tag has an `href` attribute that says where the link points; the content of the tag is the text of the link:

```
<a href="http://rci.rutgers.edu/~ag978/litdata">Course web site</a>
```

which comes out as a clickable link

Course web site

on a webpage.

Handling HTML (exercise)

The XML package is the R workhorse for probing HTML, too. The only formal differences are that you use htmlParse instead of xmlParse, and you can forget all about namespaces. However, HTML structures are typically much less intricate than XML structures, and also much more ad hoc and messy. That is because documents on the web are encoded with a primary emphasis on presentation and usability for the human browser, not on the meaningfulness of their encoding structure.

As an exercise in applying what you know about XML to HTML, download the home page for our Graduate Program, (use your browser's "Save As…" and make sure you're only saving the HTML) to the same directory as your homework. Then write code to load it in and construct an expression to store in a variable grad_ps all the p descendants (*not* immediate children) of the div with id attribute ja-content-main. If you have done this right, then

```
grad_ps[[1]]
```

will yield our program's introductory "boast."

HTML: simpler, so more complex (no exercise)

HTML has a very limited vocabulary of tags. In order to find specific content in HTML, you typically narrow down by examining the source code and paying close attention to two attributes: class and id. These attributes are what HTML designers use to structure web pages. The graphical layout of web pages is produced by a series of transformations based, mostly, on the class and id values of elements, together with their positions in the hierarchy of the document. Cascading Style Sheets (CSS), which is used to lay out the Web, does all of its design work in these terms; for example: "Make all h2 children of div elements of class blog_post bold, 18 point, sans serif font." Because web pages are structured to be styled with CSS, they can usually also be examined with XPaths. Indeed, the language of CSS selectors is very closely related to XPath. (For a reasonable introduction, see HTML Dog's tutorial. Very optional reading.)

In brief, the XPath `x//y` is written in CSS as `x y`; `x[@id="abc"]` is written `x#abc`; and `x[@class="abc"]` is written `x.abc`. The great advantage of knowing a little about CSS when you want to mine webpages is that you can make use of a tool like selectorgadget or your browser's own "DOM Inspector" function to quickly figure out what kind of path is need to pick out certain elements in the HTML.

If you want to use CSS rather than XPath, however, you cannot directly use XML's getNodeSet function (or xpathApply, or similar). Instead, you'd have to turn to the in-development package rvest. Read the vignette that comes with this package:

```
vignette(package="rvest", "selectorgadget")
```

Installing selectorgadget in your web browser is completely optional. In any case, try out the commands in the R code at the end of that vignette for yourself. (Underneath the hood of the rvest package, the XML package is still doing most of the hard work.) There is no need to include this in your submitted homework.

Downloading the Web (no exercise)

If you are after a whole bunch of files on the web, using "Save As..." by hand quickly grows tiresome. You can do this using programming, if you can figure out the URLs of the files you want to download. The `httr` package for R gives you functions for requesting a URL and extracting the results:

```
library("httr")
```

```
uri <- "http://nytimes.com"
nyt_homepage <- GET(uri)
homepage_text <- content(nyt_homepage, as="text",
                         encoding="UTF-8")
# and save the whole page to a file
writeLines(homepage_text, "nyt-frontpage.html")
```

If I wanted to parse the results later:

```
homepage_html <- htmlParse("nyt-frontpage.html",
                           encoding="UTF-8")
headlines <- xpathSApply(homepage_html,
                         "//h3[@class='story-heading']",
                         xmlValue)
headlines[1]
```

```
[1] "\n        Yemen’s War Leaves Aden Crumbling and StarvingNYT Now        "
```

or equivalently, using `rvest`:

```
library("rvest")
```

```
headlines <- homepage_html %>%
    html_nodes("h3.story-heading") %>%
    html_text()
headlines[1]
```

```
[1] "\n        Yemen’s War Leaves Aden Crumbling and StarvingNYT Now        "
```

(If you try these lines out yourself, you might, of course, see different headlines.)

With patience and `for` loops, one can write R scripts to download many web pages. It's important to know that if you are downloading in a program, you *must* enforce a delay between each download, or you will be in effect "attacking" the server hosting the web page you are asking for. The way to delay in R is to use the `Sys.sleep` function. `Sys.sleep(x)` causes R to wait for `x` seconds (`x` can be less than 1). Different servers have different tolerances for multiple requests from the same source.

However, it has to be said that if you are doing serious Web downloading, an R program is not the best choice. Instead, you could make use of the `wget` command-line program, which can do lots of neat things for you, like downloading an entire website. This is a program you must install separately and use from the Terminal (Mac) or the Command Prompt (Windows). It is fairly easy to learn to use for practiced programmers like yourselves; an excellent introduction can be found on The Programming Historian.