CMU MSP 36602: Web Scraping in R

H. Seltman, Wed. Jan. 23, 2019

1) **Basic idea:**
   a) ***Find a web page*** that has the data you want or allows access to the data you want. (Consider capacity limits on usage, and be ethical in the face of statements forbidding scraping.)
   b) If the data is on many pages each of which has a URL that depends on the specific data needed, deduce if you can just use a web address (***GET*** request) or if you need to ***POST*** your request. (More details are at https://www.diffen.com/difference/GET-vs-POST-HTTP-Requests.)
   c) If the page offers the ***data as a data file***, e.g., .sas7bdat or .json, just download it. More likely, you will need to do this repeatedly, so you just need to write a program using a mechanism to load files from the web based on their web addresses.
   d) If the page obtained through GET or POST has the ***data embedded in its html***, use tools to extract the html elements, and then extract the data in a form you can use.
   e) See shinyDesignPhase.pdf in 36612 for an intro to HTML

2) **Basic scraping in R**
   a) Example 1: a **text file** (or html as text), e.g., http://www.stat.cmu.edu/~hseltman/scrape1.txt
   R (scrape1.R):

```
url = "http://www.stat.cmu.edu/~hseltman/scrape1.txt"
text = readLines(url)
text
length(text)
text[1]
```

   b) Example 2: a **csv file**, e.g., http://www.stat.cmu.edu/~hseltman/scrape2.csv
   R (scrape2.R):

```
url = "http://www.stat.cmu.edu/~hseltman/scrape2.csv"
dtf = read.csv(url, as.is=TRUE)
str(dtf)
```

   c) Example 3: a **json file** (just Javascript Object Notation: a format, not a web protocol)
   R (stations.R):

```
library(jsonlite)  # may need to install first
url = "https://feeds.citibikenyc.com/stations/stations.json"
nycBikes = fromJSON(url)
length(nycBikes)
sapply(nycBikes, class)
sapply(nycBikes, length)
dim(nycBikes[["stationBeanList"]])
head(nycBikes[["stationBeanList"]], 4)
```

3) Intermediate scraping in R: Scraping **html elements, e.g.,** tables, lists, and links

a) Native R [used only for simple cases] (cmuPeople.R)

```
# People page of the Statistics Department
cmuPeople = "http://www.stat.cmu.edu/people"

# Read as text
people = tryCatch({
  readLines(cmuPeople, warn=FALSE)
}, warning = function(w) {
  if (grepl("HTTP status was '404 Not Found'", w$message))
    stop("web page at ", cmuPeople, " was not found")
  message(w$message, "\n")
}, error = function(e) {
  stop("Read of ", cmuPeople, " failed: ", e$message)
})

# manual examination of the HTML
cat(people[215:222], sep="\n")
#   <div class="views-field views-field-nid">
#     <span class="field-content">
#     <a href="/people/faculty/sbalakri" class="colorbox-node"
#        data-inner-width="650" data-inner-height="800"><div>
# <br>
# <img src="http://www.stat.cmu.edu/sites/default/files/faculty_pictures/siva3.jpg"
#   width="423" height="540" alt="" />
# <span class="person-caption">
# <span class="person-name">Sivaraman Balakrishnan</span><br>
# <span class="person-title">Assistant Professor</span><br>
# <span class="person-email">siva@stat.cmu.edu</span></span>
# </div></a></span>  </div>    </div>

facStart = grep("/people/faculty/", people)
cat(length(facStart), "faculty found\n")

# Extract faculty names (4 lines down) using a regular expression
facNames = gsub("(.*>)([^<]+)(.*)", "\\2", people[facStart + 4])

print(facNames[1:12])
#  [1] "Sivaraman Balakrishnan" "Anthony Brockwell"      "Olga Chilina"
#  [4] "David Choi"             "Alexandra Chouldechova" "Bernie Devlin"
#  [7] "Rebecca Doerge"         "Bill Eddy"              "Peter Freeman"
# [10] "Max G&#039;Sell"        "Christopher Genovese"   "Joel Greenhouse"

# We could write a function to convert "character entities" of the form &#039;"
# to their correct strings, but HTML also allows "&#x0027" (hex) and "&apos;".
intToUtf8(39)  # "'"
intToUtf8(0x27)  # "'"

# Convert HTML "character entities"
if (!require(xml2)) stop("load 'xml2' library and try again")
# Taken from:
# https://stackoverflow.com/questions/5060076/convert-html-
#   character-entity-encoding-in-r/35187962#35187962
unescape_html <- function(str) {
  xml2::xml_text(xml2::read_html(paste0("<x>", str, "</x>")))
}
```

```
# Test the function
unescape_html(facNames[10:12]) #  Note: only first one done

# Convert the names.  Note that without USE.NAMES=FALSE, the
# result is a named vector with the original version as the names.
facNames = sapply(facNames, unescape_html, USE.NAMES=FALSE)
facNames[1:12]
#  [1] "Sivaraman Balakrishnan" "Anthony Brockwell"      "Olga Chilina"
#  [4] "David Choi"             "Alexandra Chouldechova" "Bernie Devlin"
#  [7] "Rebecca Doerge"         "Bill Eddy"              "Peter Freeman"
# [10] "Max G'Sell"             "Christopher Genovese"   "Joel Greenhouse"
```

b) Understanding "selection" for html/css as used in the "rvest" package

You should do the exercises at http://flukeout.github.io/

i) **Basic types**:

- "myType": all tags of type "myType" (<myType ...>)
- "#myId": all tags with id="myId"
- ".myClass: all tags with class="myClass"

ii) Using a space to indicate **nesting**:

- "myType1 myType2": all tags of type "myType2" ***nested*** in type "myType1"
- "#myId myType": all tags of type "myType" ***nested*** in tags with id="myId"

iii) A class **and** a type or id:

- "#myId.myClass": all tags with class="myClass" and id="myId"

iv) Using a comma for **union**:

- ".myClass, #myId": all tags with class="myClass" or id="myId"

v) Using "*" for **all** tags:
".myClass *": all tags inside of tags with class="myClass"

vi) Using "+" for non-nested immediately subsequent **sibling** tags:

- "#myId1 + #myId2": all "sibling" tags with id="myId2" that immediately follow a tag with id="myId1"

vii) Using "~" for all non-nested subsequent "sibling" tags:

- "#myId1 ~ #myId2": all subsequent "siblings" of tags with id="myId2" that follow a tags with id="myId1"

viii) Using ">" for **immediate child** tags:

- ".myClass > #myId": within all tags with class="myClass", select children with id="myId"

ix) Using ":" as "**:first child**", ":only-child", ":last-child", "nth-child(#)":

- ".myClass *:only-child": only children within tags with class="myClass"
- "div span:nth-child(5)": "<span ...>" tags that are the 5th child of a "<div ...>"

x) Using "**:first-of-type**", ":nth-of-type()", ":last-of-type", ":only-of-type":

- "div a:first-of-type": first anchor in each div
- "div a:nth-of-type(3)": third anchor in each div
- "div a:nth-of-type(odd)": first, third, etc. anchor in each div
- "div a:only-of-type": lone anchors within divs

xi) Using "**:empty**":

- "div:empty": empty divs

xii) Using "**:not()**":

- "div:not(.myClass, #myId)": divs that have neither class="myClass" nor id="myId"

xiii) Using "**[**myAttribute**]**" (=: equals; ^=: starts with; $=: ends with; *=: contains):

- "a[width]": select anchors with any "width=" attribute
- "a[target='_blank'": select anchors with "target='_blank'"
- "a [src*='abc']": anchors with "abc" somewhere in the "src=" string

c) R, using the "rvest" package to "harvest" data (scrape3.R)

```
if (!require("rvest")) stop("load 'rvest' library and try again")
library(rvest)

# Read and parse the "scrape3" web page
url = "http://www.stat.cmu.edu/~hseltman/scrape3.html"
doc = read_html(url)

# Pull out all of the ordered lists (there should be two)
docLists = html_nodes(doc, 'ol')
if (length(docLists) != 2)
  stop("expected 2 lists, found ", length(docList))

# Pull out the first of the lists
list1 = docLists[[1]]
list1


# A function to extract the text from the lists as a vector
listContents = function(node) {
  return(trimws(html_text(html_nodes(node, "li"))))
}

# Make a vector version of the first list
list1 = listContents(list1)
list1

# Make a vector version of the second list
list2 = docLists[[2]]
```

```
list2 = listContents(list2)
list2

# Compare to a different approach
trimws(html_text(html_nodes(doc, "ol li")))

# Pull out the table (just one)
docTable = html_nodes(doc, 'table')
length(docTable) # should be 1

# Convert the one (and only) table to a data.frame
# (if any cells are missing, fill=TRUE inserts NAs)
table1 = html_table(docTable[[1]], fill=TRUE)
table1

# Pull out all hyperlinks
docAnchors = html_nodes(doc, "a")
length(docAnchors) # 1
html_text(docAnchors[1])
html_attrs(docAnchors[1])
html_attr(docAnchors[1], "href")
```

d) CMU Stats People Page with rvest

```
library(rvest)  # loads "xlm2"

doc = xml2::read_html(cmuPeople)
allPeople = rvest::html_nodes(doc, "div.views-field span.field-
content")
if (length(allPeople) == 0) stop("no faculty found")
allFaculty = NULL
for (nodeNum in seq(along=allPeople)) {
  node = allPeople[[nodeNum]]
  anchor = html_children(node)[[1]]
  if (length(anchor) == 0 || html_name(anchor) != "a") {
    # warning("no anchor in node #", nodeNum)
    next
  }
  link = html_attr(anchor, "href")
  if (is.na(link)) {
    warning("no link in #", nodeNum)
    next
  }
  if (!grepl("^/people/faculty", link)) next
  grandchild = html_children(anchor)
  if (length(grandchild) == 0 || html_name(grandchild) != "div") {
    warning("no nested div in #", nodeNum)
    next
  }
  nameNode = html_nodes(node, "div span.person-name")
  if (length(nameNode) == 0) {
    warning("no person-name in #", nodeNum)
    next
  }
  allFaculty = c(allFaculty, html_text(nameNode))
}
print(allFaculty)
```

e) Simulating **forms**

i) Example 1: GET method (statsci.R)

When you go to http://www.statsci.org/search.html, you see a form with an input box labeled "Terms" and a "Search" button. You can search for which datasets meet your criteria by entering some search info, and clicking the button. Note that this functionality is a "form" and the underlying html code is:

```
<form METHOD="GET"
ACTION="http://www.statsci.org/cgibin/searchwg.pl"
      NAME="SearchForm">
    <p><input NAME="Terms" SIZE="50" MAXLENGTH="800" VALUE>
       <input TYPE="submit" VALUE="Search"> <br>
       Example: <strong>+rats econometric -rodents</strong>
    </p>
</form>
```

If you enter "carbon or mammal" and click "Search" you end up at the url http://www.statsci.org/cgibin/searchwg.pl?Terms=carbon+or+mammal which contains links to datasets with "carbon" or "mammal". This (html-based) web page was not pre-stored on statsci's web site. Instead, it was dynamically generated by the Python cgi program "searchwg.pl" from its "standard input" value of "Terms=carbon+or+mammal". This happened because your browser constructed the url from the ACTION and the input NAME(s) and VALUE(s), and asked the server (using the "GET" method) to run the specified cgi program and return its results to the client as a webpage.

You can "simulate" asking for data through the form by constructing a url like the one the form does.

```
library(rvest)
url = "http://www.statsci.org/cgibin/searchwg.pl"
Names = "Terms"
Values = "logistic"
RHS = paste(Names, Values, sep="=", collapse="&")
doc = read_html(paste0(url, "?", RHS))
links = html_nodes(doc, 'a')
length(links)
linkList = data.frame(text = sapply(links, html_text),
                      link = sapply(links, html_attr,
                                    name="href"))
linkList
```

ii) Example 2: POST method

(1) GET (above) just specifies dynamic web pages by adding parameters to the URL. In addition to a length limit for the request (as small as 1024 characters in some browsers), all information is required to be text, is visible, is cached by the browser, and is sent unencrypted. POST does not show the request information, and that information is un-cached, has unlimited length, and may be in any binary format in addition to a text format.

(2) Example details (avcodes.R)
Link: Go to http://www.avcodes.co.uk, click on Airport Codes, to get to http://www.avcodes.co.uk/aptcodesearch.asp. Viewing the page source (and simplifying) leads to:

```
<form method="POST" action="aptcoderes.asp">
   <input type="text" name="iataapt">
   <input type="text" name="icaoapt">
   <input type="text" name="faaapt">
   <input type="text" name="locname">
   <input type="text" name="aptname">
   <select size="1" name="country">
   <OPTION VALUE='Afghanistan'>Afghanistan<OPTION ...
   </select>
   <input type="submit" value="Submit" name="B1">
   <input type="reset" value="Reset" name="B2">
</form>
```

Therefore, we need to send parameters corresponding to the "name" fields to a website corresponding to the base url (http://www.avcodes.co.uk) which is the "action" location.

```
if (!require(httr)) install.packages("httr")
library(httr)
airUrl = "http://www.avcodes.co.uk/aptcoderes.asp"
airport = POST(airUrl, body=list(iataapt='ABZ', B1='Submit'),
               encode="form")
cat(content(airport, "text"))
airDoc = read_html(airport)  # from package 'rvest'
html_children(html_children(airDoc)[2])
tds = html_nodes(airDoc, 'td')
tds
sapply(tds, html_name)
tdText = sapply(tds, html_text)
loc = grep("ICAO", tdText)
icaoCode = strsplit(tdText[loc], ":")[[1]][2]
print(icaoCode)
```