Acquiring data from the web

Teaching note:

Acquiring data is vitally important to the data scientist. The Internet is full of data but only a small fraction of it is made available in a format that is easily analyzed with statistical software. Often we will want to "mash up" data from multiple sources to produce a new analysis. For these reasons, we will often have a need to "scrape" data found on websites. We will go over some of the basic techniques for web scraping with R, and work through some examples.

Parsing a web page with R's XML library:

In this exercise, we will acquire some of the S&P 500 dataset that we used in the time series analysis tutorial, by scraping the Yahoo! Finance website. The first thing we need to do is locate the URL from which we can scrape the data.

1. Go to http://finance.yahoo.com in any web browser. Find the page for the S&P 500 index, and click on the link for "Historical Prices".

2. Do a search for monthly data from January 1, 2008, to December 31, 2012.

3. The results should all be visible on a single page. Note the URL of that page. (As of May 2013, the URL is http://finance.yahoo.com/q/hp?s=%5EGSPC&a=00&b=1&c=2008&d=11&e=31&f=2012&g=m.)

4. Now open R. We will use the `XML` package, an extension to the base R language which gives us functions for working with XML data. (The HTML language in which web pages are coded is a subset of XML. Sort of.) The `library()` command loads a package, which is presumed to already be installed on your computer, so that you may use the data and functions it contains.

   ```
   library(XML)
   ```

   If the `library()` function fails, it may be that the `XML` package is not installed on your computer. You can install the package with a line like the following:

```
install.packages("XML", dependencies = TRUE)
```

5. Use the htmlParse() function to read the web page. This will automatically create a data structure representing the entire XML "tree" in the code, making it easy for us to access in the next step. Store it in a variable called SP500.doc:

```
SP500.doc <- htmlParse("http://finance.yahoo.com/q/hp?s=%5EGSPC&a=00
&b=1&c=2008&d=11&e=31&f=2012&g=m")
```

6. The data we want is in an HTML table. The function `readHTMLTable()` returns an R list containing all the web page's tables as R data frames. We can find out how many tables are in the list by taking its length(), and we can see what sort of data they contain by using head() or str().

```
SP500.tables <- readHTMLTable(SP500.doc)
length(SP500.tables)
head(SP500.tables)
str(SP500.tables)
```

7. The output of these commands is somewhat hard to read, because web data is messy. With some trial and error, we find out that the 15th table contains the price data we're looking for. Check for yourself that this is correct, as the website design may have changed since this tutorial was prepared.

```
str(SP500.tables[[15]])
SP500.tables[[15]]
```

We specifically want the closing prices. Let's assign them to a new variable, like so:

```
SP500.close <- SP500.tables[[15]]$Close
```

Now view the dataset:

```
SP500.close  # view the dataset
```

8. There are a couple of problems remaining with this data. First, R recognizes it as "factor" data rather than numeric data. Transforming the data into numeric data is complicated by the fact that the numbers have commas in them, for example, "1,426.19". We can use the gsub() function to replace every comma with "", an empty character vector. This has the effect of transforming the data from a factor to a character type. Transform it to numeric data with as.numeric():

```
SP500.close <- gsub(",","",SP500.close)
SP500.close <- as.numeric(SP500.close)
```

Second, the last value in the dataset is an NA rather than a number. (This is due to the web page having a final row for a footnote, without any numbers in the row.) Shorten the dataset to sixty observations:

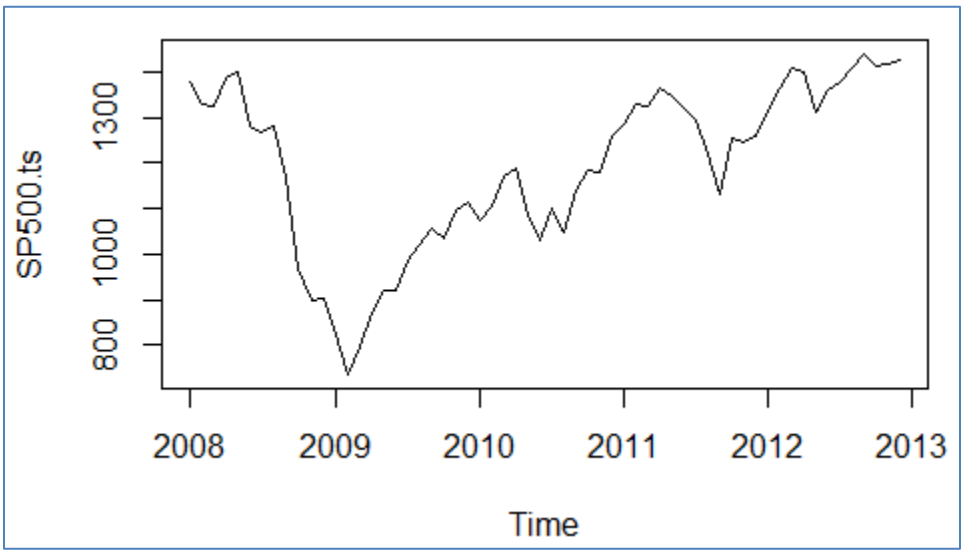
```
SP500.close <- SP500.close[1:60]
```

9. We now have a dataset we can use! We should save this data to a file on the hard disk so that we can come back to it later. We'll write it as a CSV file in the current working directory. Alternatively, you could specify a complete address for the file.

```
write.csv(SP500.close,"sp500monthly.csv")
write.csv(SP500.close,"C:\\datasets\\sp500monthly.csv")
```

We could use this data for the exercises in the time series analysis tutorial. Recall that the first steps from that tutorial were to reverse the order of the data (so it is in proper chronological order), transform it into an R time series object with `ts()`, and plot it:

```
SP500.close <- rev(SP500.close)
SP500.ts <- ts(SP500.close,start=c(2008,1),end=c(2012,12),frequency=12)
plot(SP500.ts)
```



Scraping and combining more than one page:

When the data is spread across multiple pages, we need to scrape it from each page and combine the data. Although this can be done manually, it would be painstaking work if there were more than just a few pages. We will want to automate the process. Let's figure out how to do this to acquire *weekly* S&P 500 data from January 2000 to December 2012.

1. We'll start by investigating the URLs containing the data, to see if we can figure out the pattern to them. Returning to the Yahoo! Finance page, we now specify the following search terms:

The URL for the first page of results is http://finance.yahoo.com/q/hp?s=%5EGSPC&a=00&b=1&c=2000&d=11&e=31&f=2012&g=w. One thing you need to know about URLs is that the code after the "?" is typically a list of *arguments* that specify certain options to customize what the web server sends back. Compare this URL to the one we found above. Key differences are that "2008" has become "2000" reflecting our choice of start date, and that "g=m" has changed to "g=w", apparently corresponding to the change from monthly results to weekly results.

By clicking "Next" we can see the second page of results. The arguments in its URL are: ?s=%5EGSPC&a=00&b=1&c=2000&d=11&e=31&f=2012&g=w&z=66&y=66.

Now clicking "Last" for the last page of results, we see that its URL contains: ?s=%5EGSPC&a=00&b=1&c=2000&d=11&e=31&f=2012&g=w&z=66&y=660

What's the pattern here? It may not be readily evident if you don't have much experience, but you may soon find out by trial and error and looking at additional pages of search results that the "z" parameter seems to indicate the maximum number of results per page, and the "y" parameter seems to indicate the first result on a given page. By convention, numbering starts with zero, so "y=66" means that the second page begins with the 67th data point. We can test this hypothesis by generating a page with "y=0" and see if it is identical to the original start page which had no "y" argument:

http://finance.yahoo.com/q/hp?s=%5EGSPC&a=00&b=1&c=2000&d=11&e=31&f=2012&g=w&z=66&y=0

And indeed that is the case.

2. There are 19 records on the last page, which begins with the 661st data point, which means there are 679 data points in total. This is about what we expect, given that 52×13 = 676 and that a year is just slightly longer than 52 weeks.

   You might guess that a simple solution to our multi-page problem is to change the "z" parameter to show 679 data points on a single page. Try modifying the URL to start with the first data point (#0) and show 679 records:

   http://finance.yahoo.com/q/hp?s=%5EGSPC&a=00&b=1&c=2000&d=11&e=31&f=2012&g=w&z=679&y=0

Alas, it does not work. Yahoo!'s website still shows only 66 data points per page, and there's nothing we as outsiders can do about it.

3. Instead what we'll have to do is visit each page of results in its turn and scrape the data (as we did previously), then combine the data into one big dataset. We can do this with an R loop. The only thing we need to change in each URL is the "y" value, going up from 0 to 660 in increments of 66. The expression `seq(0,679,66)` gives us all the numbers in the sequence.

```
y.values <- seq(0,679,66)
```

A simple R loop prints to the screen all the URLs we will visit. It uses `paste()` to add each "y" value in turn to the end of the URL, with `sep=""` to make sure no blank space is inserted into the URL. The first part of the URL is broken up into sections to make it fit legibly here, but you could condense it in your code.

```
for(y in y.values) {
  URL <- paste("http://finance.yahoo.com/q/hp",
               "?s=%5EGSPC&a=00&b=1&c=2000&d=11",
               "&e=31&f=2012&g=w&z=679&y=",
               as.character(y),
               sep="")
  print(URL)
}
```

If you aren't familiar with R loops, which are like loops in other programming languages, consult a good programming guide. The quick explanation is that R executes the code in between the `{ }` brackets once for every value in the vector `y.values`. In our case, that means it generates a URL ending with each of the numbers from 0 to 660 by steps of 66.

4. We extend the loop by adding the code we previously developed for the single web page example. We remove the `print()` statement and instead tell R to parse the web page, find the data we're interested in, and clean it up as before.

In order to combine all the data, we create an empty numeric vector called `SP500weekly` before the loop begins, and at each iteration we append the new values to that vector. When the loop finishes, `SP500weekly` should contain the entire dataset. It may take some time to run, because R must repeatedly access and process pages on the web site.

```
SP500weekly = numeric() # an empty numeric vector

for(y in y.values) {

  # generate the URL
  URL <- paste("http://finance.yahoo.com/q/hp",
               "?s=%5EGSPC&a=00&b=1&c=2000&d=11",
               "&e=31&f=2012&g=w&z=679&y=",
               as.character(y),
               sep="")

  # get the data from the web, as before
```

```
  SP500.doc <- htmlParse(URL)
  SP500.tables <- readHTMLTable(SP500.doc)
  SP500.close <- SP500.tables[[15]]$Close

  # clean up the data, as before
  SP500.close <- gsub(",","",SP500.close)
  SP500.close <- as.numeric(SP500.close)
  SP500.close <- SP500.close[1:(length(SP500.close)-1)] # note change

  # add this page's data to the dataset
  SP500weekly <- append(SP500weekly,SP500.close)

}
```

Note the change in one line of the "clean up" code. Since the last page does not have as many data points as the others, but we need to remove the trailing NA value from each, we tell R to cut the vector to its length minus one – regardless of what that length may be. Otherwise this code is the same as above.

5. Review the data in `SP500weekly`. There's a problem here: there are 689 records instead of the expected 679. It turns out that in every page except the first, the first data point duplicates the last price from the previous page (apparently representing weeks that have been "cut in half" by the pagination).

   To remove the duplication, we can use an `if` statement to remove the first value from the vector, conditional on this not being the first page. Amend the loop by adding the `if` statement before the line that appends the data to `SP500weekly`:

```
  # remove duplicated value in position 1
  if (y>0) { # i.e. if this is not the first "y"
    SP500.close <- SP500.close[2:length(SP500.close)]
  }
```

   Now re-run the whole loop and check the result. You should have the 679 prices we wanted.

Websites are rarely designed to make it easy for you to scrape their data, but with a little trial-and-error and a few tricks like deciphering what their URLs mean, you can gain access to a vast array of interesting data. The more you know about HTML and web servers, the more messy sites you may be able to parse.

Before scraping data from the web, carefully consider whether you could get yourself into legal trouble, or place an undue burden on the web server you're accessing. We used Yahoo! Finance as an example because they freely offer the same data for download, and as a high-traffic site they aren't likely to be affected by a few students running one-time data scraping scripts.