Digital Data Collection - Digging Deeper

Rolf Fredheim

24/02/2015

Digital Data Collection - Digging Deeper

Logging on

Before you sit down: - Do you have your MCS password? - Do you have your Raven password? - If you answered **'no'** to either then go to the University Computing Services (just outside the door) NOW! - Are you registered? If not, see me!

Download these slides

Follow link from course description on the SSRMC pages or go directly to `http://fredheir.github.io/WebScraping/`

Download the R file to your computer

Install the following packages:

ggplot2 lubridate plyr jsonlite stringr

No class next week!!

Recap

- Basic principles of data collection
- Basics of text manipulation in R
- Simple scraping example

Today we will scrape

More JSON! - social share stats - comments - newspaper articles

For that we will need

- use paste to make urls
- jsonlite to convert json to lists and data.frames
- loops to iterate over urls
- functions to store code
- rbind, cbind, and c to collect data

This might seem a lot. But very little changes, and it's a powerful toolkit

Load the packages

```
require(ggplot2)
require(lubridate)
require(plyr)
require(stringr)
require(jsonlite)
```

Last week's example

```
url  <- "http://stats.grok.se/json/en/201201/web_scraping"
raw.data <- readLines(url, warn="F")
rd  <- fromJSON(raw.data)
summary(rd)
```

```
            Length Class  Mode
daily_views 31     -none- list
project      1     -none- character
month        1     -none- character
rank         1     -none- numeric
title        1     -none- character
```

Cont

```
rd.views <- unlist(rd$daily_views )
rd.views
```

```
2012-01-01 2012-01-02 2012-01-03 2012-01-04 2012-01-05 2012-01-06
       283        573        578        666        673        626
2012-01-07 2012-01-08 2012-01-09 2012-01-24 2012-01-25 2012-01-22
       360        430        747        771        758        458
2012-01-23 2012-01-20 2012-01-21 2012-01-17 2012-01-16 2012-01-15
       673        739        536        730        669        568
2012-01-14 2012-01-13 2012-01-12 2012-01-11 2012-01-10 2012-01-29
       439        742        710        800        716        500
2012-01-31 2012-01-30 2012-01-19 2012-01-18 2012-01-26 2012-01-27
       753        838        726        734        739        738
2012-01-28
       490
```

What are the moving parts?

in url: - date - language - wiki page

in response: - field name (daily_views)

Sorting a data frame

- Use order()
- This will return ranks:
- These ranks can be applied using square bracket notation

```
df <- data.frame(rd.views)
df$dates <-rownames(df)
order(rownames(df))
```

```
 [1]  1  2  3  4  5  6  7  8  9 23 22 21 20 19 18 17 16 28 27 14 15 12 13
[24] 10 11 29 30 31 24 26 25
```

```
ord_df <-df[order(rownames(df)),]
ord_df
```

```
           rd.views      dates
2012-01-01      283 2012-01-01
2012-01-02      573 2012-01-02
2012-01-03      578 2012-01-03
2012-01-04      666 2012-01-04
```

Changing the date

```
target <- 201401
url <- paste("http://stats.grok.se/json/en/",
          target,"/web_scraping",sep="")

getData <- function(url){
    raw.data <- readLines(url, warn="F")
    rd  <- fromJSON(raw.data)
    rd.views <- unlist(rd$daily_views )
    df <- data.frame(rd.views)
  #Because row names tend to get lost....
  df$dates <- rownames(df)
    return(df)
}

getData(url)
```

```
           rd.views      dates
2014-01-15      779 2014-01-15
2014-01-14      806 2014-01-14
```

Create urls for January -June

- ':' operator
- paste()

```
5:10
```

```
[1]  5  6  7  8  9 10
```

```
201401:201406
```

```
[1] 201401 201402 201403 201404 201405 201406
```

```
targets <- 201401:201406
target_urls <- paste("http://stats.grok.se/json/en/",
                     targets,"/web_scraping",sep="")
target_urls
```

```
[1] "http://stats.grok.se/json/en/201401/web_scraping"
[2] "http://stats.grok.se/json/en/201402/web_scraping"
[3] "http://stats.grok.se/json/en/201403/web_scraping"
[4] "http://stats.grok.se/json/en/201404/web_scraping"
```

Download them one by one

```
for (i in target_urls){
    print (i)
}
```

```
[1] "http://stats.grok.se/json/en/201401/web_scraping"
[1] "http://stats.grok.se/json/en/201402/web_scraping"
[1] "http://stats.grok.se/json/en/201403/web_scraping"
[1] "http://stats.grok.se/json/en/201404/web_scraping"
[1] "http://stats.grok.se/json/en/201405/web_scraping"
[1] "http://stats.grok.se/json/en/201406/web_scraping"
```

```
for (i in target_urls){
    dat = getData(i)
}
```

Binding data together

3 functions: - c() - rbind() - cbind()

Simple solution: bind object A together with new object, and overwrite object A. Repeat.

Loops: storing the data?

- Usually we use the variable i
- Why variable? Because we can reuse 'i', even though the value it refers to 'varies'
- Why i? i is for index. But you could use anything you want

```
hold <- NULL
for (i in 1:5){
  print(paste0('this is loop number ',i))
  hold <- c(hold,i)
  print(hold)
}
```

```
[1] "this is loop number 1"
[1] 1
[1] "this is loop number 2"
[1] 1 2
[1] "this is loop number 3"
[1] 1 2 3
[1] "this is loop number 4"
[1] 1 2 3 4
```

Solution

use rbind() create empty vector, and add data to the end of it:

```
holder <- NULL
for (i in target_urls){
    dat <- getData(i)
    holder <- rbind(holder,dat)
}

holder
```

```
           rd.views      dates
2014-01-15      779 2014-01-15
2014-01-14      806 2014-01-14
2014-01-17      827 2014-01-17
2014-01-16      981 2014-01-16
2014-01-11      489 2014-01-11
2014-01-10      782 2014-01-10
2014-01-13      756 2014-01-13
2014-01-12      476 2014-01-12
```

Is this efficient?

Why (not)?

Parsimonious approach

Create a matrix and assign using square bracket notation - if you know the number of rows

You could also use ldply here: $>$ For each element of a list, apply function then combine results into a data frame.

Does the same thing as lapply + do.call(rbind) - lapply: 'applies' a function to each item in a vector. Returns a list. - do.call: executes a function to each part of an item (here: the list)

Apply family: apply(), sapply(), lapply() extensions from Hadley Wickham's plyr: ddply(), ldply() the most useful

```
dat <- ldply(target_urls,getData)
```

Putting it together

```
targets <- 201401:201406
targets <- paste("http://stats.grok.se/json/en/",
                 201401:201406,"/web_scraping",sep="")
dat <- ldply(targets,getData)
```

Task

Edit the code to download data for a different range of dates

Edit the second line to download a vector of pages, rather than dates:

```
targets <- c("Barack_Obama","United_States_elections,_2014")
```

Walkthrough

```
targets <- c("Barack_Obama","United_States_elections,_2014")
target_urls <- paste("http://stats.grok.se/json/en/201401/",targets,sep="")
results <- ldply(target_urls,getData)

#find number of rows for each:
t <- nrow(results)/length(targets)
t
```

```
[1] 31
```

```
#apply ids:
results$id <- rep(targets,each=t)
```

Moving on

Comments to newspaper articles

`http://www.dailymail.co.uk/news/article-2643770/Why-Americans-suckers-conspiracy-theories-The-country-founded-says-British-academic.html`

`http://www.dailymail.co.uk/reader-comments/p/asset/readcomments/2643770?max=10&order=desc`

Why can't we use our getData function?

Download the page

```
url <- 'http://www.dailymail.co.uk/reader-comments/p/asset/readcomments/2643770?max=1
raw.data <- readLines(url, warn="F")
rd  <- fromJSON(raw.data)

str(rd)
```

```
List of 3
 $ status : chr "success"
 $ code   : chr "200"
 $ payload:List of 9
  ..$ total                : int 373
  ..$ parentCommentsCount  : int 166
  ..$ offset               : chr "0"
  ..$ max                  : int 10
  ..$ page                 :'data.frame':   10 obs. of  14 variables:
  .. ..$ id                  : int [1:10] 55921963 55864818 55863015 55860458 5585934
  .. ..$ dateCreated         : chr [1:10] "2014-06-01T16:41:36.960Z" "2014-05-31T17:4
  .. ..$ message             : chr [1:10] "Step 1: Throw a bunch of \"conspiracy theo
  .. ..$ assetId             : int [1:10] 2643770 2643770 2643770 2643770 2643770 264
```

Digging in

find the list called 'payload'

here we get stats about the number of comments rd*payload*total

Dig furter into 'page' Here's a reasonably well formatted dataframe. We'll get rid of replies, though:

```
dat <- rd$payload$page
dat$replies <- NULL
head(dat)
```

```
        id               dateCreated
1 55921963 2014-06-01T16:41:36.960Z
2 55864818 2014-05-31T17:45:45.763Z
3 55863015 2014-05-31T17:10:40.931Z
4 55860458 2014-05-31T16:21:58.323Z
5 55859344 2014-05-31T15:59:16.771Z
6 55856766 2014-05-31T15:07:57.775Z
```

1
2
3

Movable parts

url <- 'http://www.dailymail.co.uk/reader-comments/p/asset/readcomments/2643770?max=10&order=desc'

- id <- 2643770
- max <- 10
- order <- desc

Try repeating the process to download 100 comments

APIs

When used in the context of web development, an API is typically defined as a set of Hypertext Transfer Protocol (HTTP) request messages, along with a definition of the structure of response messages, which is usually in an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

The practice of publishing APIs has allowed web communities to create an open architecture for sharing content and data between communities and applications. In this way, content that is created in one place can be dynamically posted and updated in multiple locations on the web

-Wikipedia

Social shares

Most newssites will give stats about social shares. E.g:
`http://www.bbc.co.uk/sport/0/football/31583092`

-> as of writing this up it had been shared 92 times

Uses Twitter and Facebook apis Work with a 'get' request

Types of request

> *GET requests a representation of the specified resource. Note that GET should not be used for operations that cause side-effects, such as using it for taking actions in web applications. One reason for this is that GET may be used arbitrarily by robots or crawlers, which should not need to consider the side effects that a request should cause.*
>
> *POST submits data to be processed (e.g., from an HTML form) to the identified resource. The data is included in the body of the request. This may result in the creation of a new resource or the updates of existing resources or both.*

we use 'get' for scraping, 'post' is more complicated. Use it to navigate logins, popups, etc.

Constructing a query

You might have seen urls with these signs in them: - ? - & The question mark indicates the start of a query, while & is used to separate fields.

Get requests to social shares are very simple:

`http://graph.facebook.com/?id=http://www.bbc.co.uk/sport/0/football/31583092`

`http://urls.api.twitter.com/1/urls/count.json?url=http://www.bbc.co.uk/sport/0/football/31583092`

Download these into R!

Facebook

```
url <- 'http://graph.facebook.com/?id=http://www.bbc.co.uk/sport/0/football/31583092'
raw.data <- readLines(url, warn="F")
rd  <- fromJSON(raw.data)
df <- data.frame(rd)
```

Repeat for Twitter

Task

1. Download the number of Twitter shares for each of these these pages:

   - `http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.html`
   - `http://www.dailymail.co.uk/news/article-2643770/Why-Americans-suckers-conspiracy-theories-The-country-founded-says-British-acader html`

2. Use rbind to combine these responses into a single data.frame
3. write a function that takes an input url to scrape Twitter
4. Write a function that takes an input url to scrape both Twitter and Facebook (hard!)
5. use ldply to make a scraper (copy code from slide 14 - parsimonious approach - above)

Walkthrough

```
#1)
url <- 'http://www.dailymail.co.uk/news/article-2643770/Why-Americans-suckers-conspir
target <- paste('http://urls.api.twitter.com/1/urls/count.json?url=',url,sep="")
raw.data <- readLines(target, warn="F")
rd  <- fromJSON(raw.data)
tw1 <- data.frame(rd)

url2 <- 'http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.h
target <- paste('http://urls.api.twitter.com/1/urls/count.json?url=',url2,sep="")
raw.data <- readLines(target, warn="F")
rd  <- fromJSON(raw.data)
tw2 <- data.frame(rd)
```

Walkthrough 2 and 3

```
#2)
df <- rbind(tw1,tw2)

#3)
getTweetCount <-function(url){
    target <- paste('http://urls.api.twitter.com/1/urls/count.json?url=',url,sep="")
    raw.data <- readLines(target, warn="F")
    rd  <- fromJSON(raw.data)
    tw1 <- data.frame(rd)
    return(tw1)
}
getTweetCount(url2)
```

```
  count
1   250
                                                                                  url
1 http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.html/
```

Walkthrough 4

```
#4)
getBoth <-function(url){
    target <- paste('http://urls.api.twitter.com/1/urls/count.json?url=',url,sep="")
    raw.data <- readLines(target, warn="F")
    rd  <- fromJSON(raw.data)
    tw1 <- data.frame(rd)

    target <- paste('http://graph.facebook.com/?id=',url,sep='')
    raw.data <- readLines(target, warn="F")
    rd  <- fromJSON(raw.data)
    fb1 <- data.frame(rd)

    df <- cbind(fb1[,1:2],tw1$count)
    colnames(df) <- c('id','fb_shares','tw_shares')
    return(df)
}
```

Walkthrough 5

```
#5)
targets <- c(
'http://www.dailymail.co.uk/news/article-2643770/Why-Americans-suckers-conspiracy-the
'http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.html'
)

dat <- ldply(targets,getBoth)
```

Comments

It's almost the same thing

```
url <- 'http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.ht
api <- 'http://graph.facebook.com/comments?id='
target <- paste(api,url,sep="")
raw.data <- readLines(target, warn="F")
rd  <- fromJSON(raw.data)
head(rd$data)
```

Getting article content

In TWO WEEKS TIME (10 March) we will do scraping proper. But check out this awesome API: `http://juicer.herokuapp.com/`

`http://juicer.herokuapp.com/api/article?url=http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.html`

Download the page

```
url <- 'http://www.huffingtonpost.com/2015/02/22/wisconsin-right-to-work_n_6731064.ht
api <- 'http://juicer.herokuapp.com/api/article?url='

target <- paste(api,url,sep="")
target

raw.data <- readLines(target, warn="F")
rd  <- fromJSON(raw.data)

dat <- rd$article
dat$entities <-NULL

dat <-data.frame(dat)
dat
```

=====

```
ent <- rd$article$entities
ent
```

What does the last frame give us?

Named entity recognition!

```
#use square bracket notation to navigate these data:
ent[ent$type=='Location',]
ent[ent$type=='Person',]
```

Summing up

given URLs of target pages, we can now: - download raw JSON data - extract fields of interest - put this in a function - apply the function to a list of targets

No class next week!!

So that's it for APIs and JSON But for those who are keen, more advanced stuff involving APIs and JSON sources (maps? YouTube?) can be found in last year's slides:

- `http://fredheir.github.io/WebScraping/Lecture4/p4.html`

Too many loops, variables, and functions?

If this has all been a bit much, below is a link to some extra material on all things variables, functions and loops

- `http://fredheir.github.io/WebScraping/Lecture2_2015/extra.html`
- `http://fredheir.github.io/WebScraping/Lecture2_2015/extra.R`