STAT243 ps2

Zhaochen(Winnie) Gao 3034268813

9/8/2018

Problem

1. Please read Unit 5 on good programming/project practices and incorporate what you've learned from that reading into your solutions (particularly for Problem 3). As your response to this question, briefly (a few sentences) note what you did in your code that reflects the material in Unit 5. You could also note anything in Unit 5 that you disagree with, if you have a different stylistic perspective.

Answer: After reading Unit 5 on good programming/project practices, I learn about the decent coding styles. Especially for problem 3, I use indentation to help me detect errors. For while loop and if statement, indentation can expose the lack of symmetry. I use variables for most cases and try not to hard code numbers and urls. I also add a lots of comments, so that I can summarize different blocks of code. For part 3d, I break the function into two smaller functions, so that I can debug more efficiently. And the first function is reusable. For functions built in problem 3, I use assertthat(), testthat() and if() combined with stop() to check the arguments provided by users.

2.This problem explores file sizes and file-reading speed and their relationship to the file format in light of understanding storage in ASCII plain text versus binary formats and the fact that numbers are stored as 8 bytes per number in binary format.

(a) Explain the sizes of the various files. In discussing the CSV text file, how many characters do you expect to be in the file (i.e., you should be able to estimate this very accurately without using wc or any explicit program that counts characters).

```
# n <- 1e7
# a <- matrix(rnorm(n), ncol = 100)
# a <- round(a, 10)
# write.table(a, file = '/tmp/tmp.csv', quote=FALSE, row.names=FALSE,
# col.names = FALSE, sep=',')
# save(a, file = '/tmp/tmp.Rda', compress = FALSE)
# file.size('/tmp/tmp.csv')
# ## [1] 133887710
# file.size('/tmp/tmp.Rda')
# ## [1] 80000087
```

Answer: If you save the table as a csv file, then default encoding is ASCII. But for Rda file, the default encoding is binary. In ASCII format, it usually takes 12 bytes for a 10-decimal-place number, 1 byte for delimiter, 1 byte for backspace for each line, 1 byte for each negative sign for some random generated numbers. So the total file size is the sum of 120000000 for numbers part, 10000000 for comma and backspace, and parts for negative signs. But for binary file, it stores information as sequence of bytes. Each number would take the exact size of 8 bytes. That's why file size of Rda is around 10000000*8.

(b) Now consider saving out the numbers one row per number. Given we no longer have to save all the commas, why is the file size unchanged?

```
# b <- a
# dim(b) <- c(1e7, 1) ## change to one column by adjusting attribute
# write.table(b, file = '/tmp/tmp-onecolumn.csv', quote=FALSE,
# row.names=FALSE, col.names = FALSE, sep=',')
# file.size('/tmp/tmp-onecolumn.csv')
# ## [1] 133887710
```

Answer: The reason why the file size remains unchanged is that comma and backspace take exactly same space in ASCII format. So if you replace comma with backspace, the file size would not change.

(c) Consider the following ways of reading the data into R. Explain the difference in speed between the two situations in "First comparison", then for the difference in "Second comparison", and for "Third comparison". Side note: in this case read_csv() is rather faster than read.csv().

```
# First comparison
# system.time(a0 <- read.csv('/tmp/tmp.csv', header = FALSE))
# ## user system elapsed
# ## 35.348 0.248 35.599
# system.time(a1 <- scan('/tmp/tmp.csv', sep = ','))
# ## user system elapsed
# ## 5.236 0.024 5.261
```

Answer: For first comparison, when we use read.csv, all columns are read as character columns and then converted using type.convert to logical, integer, numeric, complex or (depending on as.is) factor as appropriate. But when we use scan function, it will return a list or a vector, which is not a dataframe. And it will just scan the data type and simply read in.

```
# Second comparison
# system.time(a0 <- read.csv('/tmp/tmp.csv',header = FALSE, colClasses = 'numeric'))
# ## user system elapsed
# ## 5.236 0.044 5.281
# system.time(a1 <- scan('/tmp/tmp.csv', sep = ','))
# ## user system elapsed
# ## 5.256 0.032 5.289
```

Answer: When we specify the classes for each column, it will help read.csv function save huge time because R does not have to go through the process like: firstly, reading in each column as character type, then determining the type of original column and converting the column to the desired type. Using colClasses allows the desired class to be set for each column in the input. Since scan function does not set "what" argument, which specifies a list of modes of variables to be read from the file. It still needs some time for this step. The difference in speed is not as big as before.

```
# Third comparison
# system.time(a1 <- scan('/tmp/tmp.csv', sep = ','))
# ## user system elapsed
# ## 5.288 0.020 5.307
# system.time(load('/tmp/tmp.Rda'))
# ## user system elapsed
# ## 0.076 0.008 0.085
```

Answer: First of all, the file size of csv format vs Rda is not the same. csv file is much larger than Rda file.

Secondly, load() function is used for reloading the datasets written with the function save(). It restores the saved values to the current environment which should be pretty fast.

(d) Explain why tmp.Rda is so much bigger than tmp2.Rda given they both contain the same number of numeric values.

```
# save(a, file = '/tmp/tmp1.Rda')
# file.size('/tmp/tmp1.Rda')
# ## [1] 76778730
# b <- rep(rnorm(1), 1e7)
# save(b, file = '/tmp/tmp2.Rda')
# file.size('/tmp/tmp2.Rda')
# ## [1] 116494
```

Answer: a is saved as a matrix. But b uses rep() function and save the output as a numeric vector. For rep() function, if there are replications, the replications will be compressed which will help reduce the file size significantly.

3. Go to https://scholar.google.com and enter the name (including first name to help with disambiguation) for a researcher whose work interests you. (If you want to do the one that will match the problem set solutions, you can use "Trevor Hastie", who is a well-known statistician and machine learning researcher.) If you've entered the name of a researcher that Google Scholar recognizes as having a Google Scholar profile, you should see that the first item returned is a "User profile". Next, if you click on the hyperlink for the researcher under "User profiles for ", you'll see that brings you to a page that provides the citations for all of the researcher's papers. Note: if you repeatedly query the Google Scholar site too quickly, Google will start returning "503" errors because it detects automated usage (see problem 4 below). So, if you are going to run code from a script such that multiple queries would get done in quick succession, please put something like Sys.sleep(2) in between the calls that do the HTTP requests. Also when developing your code, once you have the code in part (a) working to download the HTML, use the downloaded HTML to develop the remainder of your code and don't keep re-downloading the HTML as you work on the remainder of the code.

(a) Now, based on the information returned by your work above, including the various URLs that your searching and clicking generated, write R code that will programmatically return the citation page for your researcher. Specifically, write a function whose input is the character string of the name of the researcher and whose output is the html text corresponding to the researcher's citation page as well as the researcher's Google Scholar ID.

```
#use rvest package
library(rvest)
```

```
## Loading required package: xml2
```

```
library(assertthat)
library(testthat)
```

```
cite_pg = function(scholar) {
  #if input argument is not a character string, then return error message
  if (assert_that(is.character(scholar))){
    baseurl = "https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&q="
    # substitute the blank between first name and last name as plus sign, so we can match the format of
    scholar_m = sub(" ", "+", scholar)
    # combine baseurl and scholar_m together
    url_m = paste0(baseurl, scholar_m)

    download.file(url_m, destfile = "tmp.html", quiet=TRUE)
    # read in url
    Sys.sleep(2)
    main_pg = read_html("tmp.html")
    # avoid the situation that Google return 503 error
    Sys.sleep(2)
    # find the link that directs to User Profile
    cite_link = main_pg %>% html_nodes(".gs_rt2 a") %>% html_attr('href')
    Sys.sleep(2)
    # if we cannot find the User Profile for this scholar, then return error message
    if (length(cite_link) == 0) {
      return("Google scholar cannot find this scholar, please try another one.")
    }
    # substitute part of User Profile's link to match expected url
    cite_link = sub("&oe=ASCII", "", cite_link)

    cite_url = "https://scholar.google.com"
    cite_page = paste0(cite_url, cite_link)
    return(cite_page)
  }
}
```

(b) Create a second function to process the resulting HTML to create an R data frame that contains the article title, authors, journal information, year of publication, and number of citations as five columns of information. Try your function on a second researcher to provide more confidence that your function is working properly.

```
cite_sumtab = function(scholar) {

  # take the html text for the citation page
  cite_link = cite_pg(scholar)

  # if the value of cite_link is error message, then return the error message
  if(length(grep("http", cite_link))==0){
    return(cite_link)
  }

  # read in the citation page
  main_pg = read_html(cite_pg(scholar))
  # extract the text from class gsc_a_at, which contains info for article title
  article_title = main_pg %>% html_nodes(".gsc_a_at") %>% html_text()
  # extract the text from class gs_gray, which contains info for authors, journal info and publication
  listings = main_pg %>% html_nodes(".gs_gray") %>% html_text()
```

```
  # authors info is located at odd position
  authors = listings[seq(length(listings)) %% 2 == 1]
  # combined info for journal info and publication year is located at even position
  combined = listings[seq(length(listings)) %% 2 == 0]
  # find the position of the comma before publication year
  pos = sapply(gregexpr(",", combined), tail, 1)
  # create the substring for journal information
  journal_info = substr(combined, 1, pos - 1)
  # create the substring for publication year
  pub_year = substr(combined, pos + 2, pos + 5)

  # extract the text from class gsc_a_at, which contains info for number of total citation
  cite_text = main_pg %>% html_nodes(".gsc_a_c") %>% html_text()
  cite_num = cite_text[3:length(cite_text)]

  # combine required info together and form a data frame
  cite_data = data.frame(
    article_title = article_title,
    authors = authors,
    journal_info = journal_info,
    pub_year = pub_year,
    cite_num = cite_num
    )
  return(cite_data)
}
```

(c) Based on the discussion in lab on Tuesday Sept. 4, include checks in your code so that it fails gracefully if the user provides invalid input or Google Scholar doesn't return a result. You don't have to use the assertthat package for all of your checks but please use it for at least one. Also write some test code that uses the testthat package to carry out a small number of tests of your function.

```
library(testthat)
test_that("cite_pg works with normal input", {
  x="Trevor Darrell"
  y="https://scholar.google.com/citations?user=bh-uRFMAAAAJ&hl=en&oi=ao"

  #check if citation page link got from function is matched to the correct url
  expect_equal(cite_pg(x),y)
  #check if output of the function is character string
  expect_type(cite_pg(x),typeof(y))
})

test_that("cite_pg works with invalid input", {
  x=2
  #check if we get error message with invalid input
  expect_error(cite_pg(2), "scholar is not a character vector")
})

test_that("cite_pg works with situation when Google Scholar cannot return result", {
  x="blahblah"
  #check if we get error message when Google Scholar cannot find this scholar
```

```
  expect_equal(cite_pg(x), "Google scholar cannot find this scholar, please try another one.")
})

#test for 3b function cite_sumtab()
library(assertthat)
#check if output of function is data frame
assert_that(is.data.frame(cite_sumtab("Trevor Hastie")))
```

```
## [1] TRUE
```

```
#check if output of function has 20 rows and 5 columns
test_that("cite_sumtab works with normal input",{
  x="Trevor Hastie"
  sumtab = cite_sumtab(x)
  expect_equal(nrow(sumtab),20)
  expect_equal(ncol(sumtab),5)
})

test_that("cite_sumtab works with invalid input", {
  x=2
  #check if we get error message with invalid input
  expect_error(cite_sumtab(2), "scholar is not a character vector")
})

test_that("cite_sumtab works with situation when Google Scholar cannot return result", {
  x="blahblah"
  #check if we get error message when Google Scholar cannot find this scholar
  expect_equal(cite_sumtab(x), "Google scholar cannot find this scholar, please try another one.")
})
```

(d) (Extra credit) Fix your function so that you get all of the results for a researcher and not just the first 20. Hint: figure out what query is made when you click on "Show More" at the bottom of the page. Hint: as you are trying to understand the structure of the HTML, one option is to use write_xml() on the result of read_html() and then view the file that is produced in a text editor. Note: For simplicity you can either assume only one User Profile will be returned by your initial search or that you can just use the first of the profiles.

```
cite_step = function(scholar, start_item, pg_size){

  # get the citation page url
  basic_url = cite_pg(scholar)
  # if we get error message from function cite_pg(), then return that error message
  if(length(grep("http",basic_url))==0){
    return(basic_url)
  }
  # create a new url which can help us expand the search results
  up_link = paste0(basic_url, "&cstart=", start_item, "&pagesize=", pg_size)
  # read in the html
  download.file(up_link, destfile = "tmp2.html", quiet=TRUE)
  Sys.sleep(2)
  main_pg = read_html("tmp2.html")
  Sys.sleep(2)
```

```
  # extract the text from class gsc_a_at, which contains info for article title
  article_title = main_pg %>% html_nodes(".gsc_a_at") %>% html_text()
  # if we reach the maximum number of search results, then return NULL
  if(length(article_title)==0){
    return(NULL)
  }
  Sys.sleep(2)

  # extract the text from class gs_gray, which contains info for authors, journal info and publication
  listings = main_pg %>% html_nodes(".gs_gray") %>% html_text()
  # authors info is located at odd position
  authors = listings[seq(length(listings)) %% 2 == 1]
  # combined info for journal info and publication year is located at even position
  combined = listings[seq(length(listings)) %% 2 == 0]
  # find the position of the comma before publication year
  pos = sapply(gregexpr(",", combined), tail, 1)
  # extract the substring for journal information and publication year
  journal_info = substr(combined, 1, pos - 1)
  pub_year = substr(combined, pos + 2, pos + 5)

  Sys.sleep(2)
  # extract the text from class gsc_a_at, which contains info for number of total citation
  cite_text = main_pg %>% html_nodes(".gsc_a_c") %>% html_text()
  cite_num = cite_text[3:length(cite_text)]

  # combine required info together and form a data frame
  cite_data = data.frame(
    article_title = article_title,
    authors = authors,
    journal_info = journal_info,
    pub_year = pub_year,
    cite_num = cite_num
  )
  return(cite_data)
}

cite_all = function(scholar) {
  # initialize start item and page size
  start_item = 0;
  pg_size = 20;

  base_data = cite_step(scholar, start_item, pg_size)
  # if we get error message from cite_step(), then return that error message
  if(!is.data.frame(base_data)){
    return(base_data)
  }
  # initialize sum_data
  sum_data = base_data

  # as long as we do not reach the maximum searching results, we keep expanding the search results
  while(!is.null(dim(base_data))){
    # since we already set sum_data to be base_data, we don't want to combine it again
    if(start_item!=0){
```

```
      # keep append the next 20 search results to the summary table
      sum_data = rbind(sum_data, base_data)
    }
    # everytime we increase the start_item by 20, so it can fetch next 20 search results
    start_item = start_item + 20
    # save the next 20 search results as updated base_data
    base_data = cite_step(scholar, start_item, pg_size)
  }
  # write out the generated summary table as csv file for checking
  write.csv(sum_data, "summary.csv")
  return(sum_data)
}
```

4. Look at the robots.txt file for Google Scholar and the references in Unit 3 on the ethics of webscraping. Write a paragraph or two discussing the ethics of scraping data from Google Scholar as we do in Problem 3. Do you have any concerns in terms of whether anything we are doing in Problem 3 might violate Google's rules or the general ethical considerations in the references? (I think what we are doing is in the spirit of ethical webscraping, or I wouldn't have assigned it, but there is a grey zone here, and I'd like you to think about the ethical issues within a specific context.)

Answer: In general, I think what we're doing in Problem 3 is ethical since it's just for personal use. According to robots.txt file for Google Scholar, web scraping for "/citations?user=" is allowed, which means that we are allowed to bot access the user profiles. The function we design only performs data collection, requests data at a reasonable rate and doesn't voilate the intellectual property. We are not trying to save it as our own data. In problem 3d, we are trying to download the htmls and extract all of the possible results for a researcher. Although robots.txt doesn't make statements on this behavior, we may exceed upper limit set for webscraping results. I think, to follow the spirit of ethical webscraping, if there's way to state our purpose and let Google Scholar know we build this function just for data collection, it may be better.