Stat243: Problem Set 3, Due Wednesday Sept. 30

September 18, 2015

This covers Unit 4, sections 1-4 and Unit 5 (including the R debugging tutorial).

It's due **on paper** and submitted via Github at the start of class on Sep. 30, with the additional requirement noted below for Problem 1 being due Sep. 28 at 8 am.

Some general guidelines on how to present your problem set solutions:

1. Please use your Rtex/Rnw/Rmd solution from PS1, problem 3 as your template for how to format your solutions (only non-Statistics students are allowed to use R Markdown).
2. As usual, your solution should mix textual description of your solution, code, and example output. And your code should be commented.
3. Your paper submission should be the printout of the PDF produced from your Rtex/Rnw/Rmd file. Your Github submission should include the Rtex/Rnw/Rmd file, any R or bash code files containing chunks that you read into your Rtex/Rnw/Rmd file, and the final PDF.
4. Use functions as much as possible, in particular for any repeated tasks. We will grade in part based on the modularity of your code and your use of functions.
5. Please note my comments in the syllabus about when to ask for help and about working together.
6. Please give the names of any other students that you worked with on the problem set.

Problems

1. On Monday, September 28, section will consist of a discussion of good practices in scientific computing, led by Harold and Chris. In preparation, please do the following:
   (a) Read any of Unit 5 and the *R debugging* tutorial that you haven't already looked through.
   (b) Read one of these three articles
      i. Gentzkow and Shapiro (http://www.brown.edu/Research/Shapiro/pdfs/CodeAndData.pdf)
      ii. Wilson et.al. (http://arxiv.org/pdf/1210.0530v3.pdf)
      iii. Millman and Perez (https://github.com/berkeley-stat243/stat243-fall-2014/blob/master/section/millman-perez.pdf)
   (c) Please write down a comment or question you have that is raised by these papers. Write your comment/question in a plain text file called *bestpractices.txt*, put it in the top-level directory of your Git repository, and push to Github as you do for your problem set solutions. **Please do this by 8 am Monday Sep. 28** as we'll be collating them then. In addition, include your comment/question here in your problem set solution.

(d) When reading, please think about the following questions, which we will be discussing in section:

i. Are there practices suggested that seem particularly compelling to you? What about ones that don't seem compelling to you?

ii. Do you currently use any of the practices described? Which ones, and why? Which ones do you not use and why (apart from just not being aware of them)?

iii. Why don't researchers consistently utilize these principles/tools? Which ones might be the most/least used? Which ones might be the easiest/most difficult to implement?

iv. What principles and practices described apply more to analyses of data, and which apply more to software engineering? Which principles and practices apply to both?

2. The goal of Problem 2 is two-fold: first to give you practice with regular expressions, processing HTML, and text manipulation and the second to have you thinking about writing well-structured, readable code. Regarding the latter, please focus your attention on writing short, modular functions that operate in a vectorized manner and also making use of *apply()/lapply()/sapply()* to apply functions to your data structures. Think carefully about how to structure your objects to store the debate information. You might have each candidate's response to a question be an element in a character vector or in a list.

   The website Commission on Presidential Debates has the text from recent debates between the candidates for President of the United States. (As a bit of background for those of you not familiar with the US political system, there are usually three debates between the Republican and Democratic candidates at which they are asked questions so that US voters can determine which candidate they would like to vote for.) Your task is to process the information and produce data on the debates. Note that while I present the problem below as subparts (a)-(g), your solution does not need to be divided into subparts in the same way, but you do need to make clear in your solution where and how you are doing what. Your solution should do all of the downloading and processing from within R so that your operations are self-contained and reproducible. I'm expecting your solution to be a mix of HTML processing using tools from the XML package, string processing, and use of regular expressions. For the purposes of this problem, please work on the first of the three debates from each of the years 1996, 2000, 2004, 2008, 2012. I'll call each individual response by a candidate to a question a "chunk". A chunk might just be a few words or might be multiple paragraphs. The result of all of this activity in parts (c)-(f) should be well-structured data object(s) containing the information about the debates and candidates.

   (a) From the main website, you need to get the HTML file for each of the debates. You'll need to start with the HTML file linked to above and extract the individual URLs for each speech. Then use that information to read each speech into R. Try to figure out programmatically which are the debates in the relevant time frame and which are the first of the three debates between the presidential candidates.

   (b) Extract the body of the debate for each debate. The end result of your processing is that if you use the *cat()* function on the text for a single debate, it should print out the spoken text of the speech in a nicely-formatted manner.

   (c) Convert the text so that for each debate, the spoken words are split up into individual chunks of text spoken by each speaker (including the moderator). If there are two chunks in a row spoken by a candidate, it's best to combine them into a single chunk. Make sure that any formatting and non-spoken text (e.g., the tags for 'Laughter' and 'Applause') is stripped out. There should

be some sort of metadata or attributes so that you can easily extract only the chunks for one candidate. For the Laughter and Applause tags, retain information about the number of times it occurred in the debate for each candidate. You may need to do some looping as you manipulate the text to get the chunks, but try to do as much as possible in a vectorized way.

(d) Use regular expression processing to extract the sentences and individual words as character vectors, one element per sentence and one element per word.

(e) For each candidate, for each debate, count the number of words and characters and compute the average word length for each candidate. Store this information in an R data structure. Comment briefly on the results.

(f) For each candidate, count the following words or word stems and store in an R data structure: I, we, America{,n}, democra{cy,tic}, republic, Democrat{,ic}, Republican, free{,dom}, war, God [not including God bless], God Bless, {Jesus, Christ, Christian}. Comment briefly on the results.

(g) (Extra credit) We may give extra credit for particularly nice solutions.

Hint: Depending on how your process the text, you may end up with lists for which the name of a list element is very long. Syntax such as `names(myObj) <- NULL` may be helpful.

3. Object-oriented programming practice. Consider a discrete random walk in two dimensions. At each step there is a 0.25 probability of moving left, right, up and down. An example of a random walk of three steps would be the first step is to move one unit to the right, the second step is to move one unit up, and the third step is to move one unit back down to the position attained after the first step.

   (a) Write a function that generates such a random walk. The input argument should be the number of steps to be taken. There should be an optional second argument (with a default) specifying whether the user wants the full path of the walk returned or just the final position. If the former, the result should be given in a reasonable format. As practice with defensive programming, your code should check that the input is a valid integer (it should handle zero and negative numbers and non-integers gracefully using techniques discussed in the *R debugging* tutorial).

   (b) If you didn't already, vectorize your code so that you don't need to have a for loop iterate over the steps. The *cumsum* function in particular may be helpful.

   (c) Now embed your function in object-oriented code that nicely packages up the functionality. You can choose S3, S4, or Reference Classes, but if you already have a fair amount of experience with S3, you should try one of the other two. You should create a class, called *rw*, with a constructor, a *print* method (for which the result should focus on where the final position is and some useful summary measures of the walk), a *plot* method, and a '*[*' operator than gives the position for the ith step. Also, create a replacement method called *start* that translates the origin of the random walk, e.g., `start(myWalk) <- c(5, 7)` should move the origin and the entire walk so that it starts at the position *x=5, y=7*.