STAT243 ps3

Zhaochen(Winnie) Gao 3034268813

9/18/2018

Problem 1

(a) Please write down a substantive comment or question you have that is raised by this material and submit to this Google form: by the end of the day, Monday Sep. 24 as we'll be collating them before section. In addition, include your comment/question here in your problem set solution.

Answer: 1. Through the readings, I learn about many software development practices which can improve the program's productivity and reliability. I totally agree with the statement from "Best Practices for Scientific Computing", "a program should not require its reader to hold more than a handful of facts in memory at once". Actually I'm trying to implementing some practices mentioned in the paper right now. For example, I try to pick the meaningful names for each function I built. So when others use them, they can understand the purpose of functions directly from their names. I'm also trying to avoid hard codes. So every piece of data have a single authoritative representation in the system. If I want to change the value of a particular variable, I only need to change one place. I think the place I would make self-improvements is the defensive programming. I need to add assertions to check if there's something wrong with my codes, which is helpful for debugging. Moreover, I will try to use Version Control in the future. But I think for languages like R, Python and C++, documentation may be a tricky task. Perhaps the only way to solve it is to save different versions of codes and add comments about the changes every time we make it. The paper mentions about TDD(test-driven development) and I think it's too compelling for me. Because I don't think it's possible to write all of the test cases even before writing the codes. Sometimes it's just so hard to test something that does not even exist. The other practice I think it's compelling is the suggestion that scientists should always write code in the highest-level language possible. Different languages have different advantages. Sometimes it's much easier to use lower-level languages to do visualization. And it's time-consuming to shift the codes from the higher-level to the lower-level.

(d) Based on our discussion in section, please list a few strengths and a few weaknesses of the reproducibility materials provided for the analysis in clm.pdf.

Answer: **1)** The strengths of the reproducibility materials provided for the analysis in clm.pdf includes: 1. The files have decriptive names and the description file provided is pretty clear and easy to follow. 2. The R codes are well-documented. It has different styles of comments to break up the code chunks. The indentation and spacing are pretty good as well. So they have decent formats. 3. It also provides the workflow diagram. With the help of diagram and description file, we can understand the research process more efficiently. 3. For the raw files downloading part, the codes check if files are already existed. So they allow people either to reproduce the whole thing or start in the middle. 4. Instead of combining functions in one file, this research also break them up. **2)** The weaknesses in clm.pdf includes: 1. For further usage, users have to change the file path based on their own computer systerm since the codes are using absolute path. 2. Since we need to manipulate urls, it would be better if we have codes to check if url exists. 3. The codes are not modular. If we only need the outputs for one specific part, we cannot easily extract the functions used to produce them. 4. It would be better if there are comments about summary of environment, especially the version of packages at the beginning of each file. 5. There's no testings in the R files at all.

Problem 2

(a) 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. Please print out or plot the number of chunks for the candidates.

```r
library(stringr)
library(testthat)
#remove the replicated part of debates in 2018
x = strsplit(debates_body[[3]], '\n')
debates_body[[3]] = x[[1]][4]

generate_text <- function(x) {
  #initialize lists storing text chunks, number of applause tag, number of laughter tag
  sep_lines <- app_tag <- lau_tag <- rep(list(list()), length(x))
  for (i in 1:length(x)) {
    tmp_list = list()
    #find the starting of each speech text chunk
    matches = sapply(x[i], gregexpr, pattern = '[[:upper:]]{4,12}:')
    #extract the info about candidates and assign it to the name of lists
    speaker_gp = tail(unique(regmatches(x[i], matches)[[1]]), 3)
    speaker_gp_m = sub(':', '', speaker_gp)
    sep_lines[[i]] <- app_tag[[i]] <- lau_tag[[i]] <- vector('list', length(speaker_gp))
    names(sep_lines[[i]]) <- names(app_tag[[i]]) <- names(lau_tag[[i]]) <- speaker_gp_m

    #locate each speech text chunk and store it in a temporary list
    for (j in 1:(length(matches[[1]]) - 1)) {
      tmp_list[j] = sapply(x[i], substr, matches[[1]][j], matches[[1]][j + 1] - 1)
    }
    tmp_list[length(matches[[1]])] = sapply(x[i], substr, matches[[1]][length(matches[[1]])],
                                            nchar(x[1]))

    for (m in 1:length(speaker_gp)) {
      #group elements in temp list by candidates and copy them to speech_lines
      sep_lines[[i]][speaker_gp_m[m]] = list(tmp_list[grep(speaker_gp[m], tmp_list)])

      #count number of applause and laughter occured for each candidate
      app_ct = length(grep('applause', sep_lines[[i]][[speaker_gp_m[m]]], ignore.case = TRUE))
      lau_ct = length(grep('Laughter', sep_lines[[i]][[speaker_gp_m[m]]], ignore.case = TRUE))
      app_tag[[i]][speaker_gp_m[m]] = app_ct
      lau_tag[[i]][speaker_gp_m[m]] = lau_ct

      #remove all of tags
      sep_lines[[i]][[speaker_gp_m[m]]] = sapply(sep_lines[[i]][[speaker_gp_m[m]]],
                                                  str_replace_all,
                                                  pattern = '[(|\\[][[:alpha:]]{8,}[)|\\]]',
                                                  replacement = '')

    }
  }
  result = list(sep_lines, app_tag, lau_tag)
  return(result)
}

speech_lines = generate_text(debates_body)[[1]]
applause_tag = generate_text(debates_body)[[2]]
laugh_tag = generate_text(debates_body)[[3]]


test_that("generate_text works with normal input", {
  x = generate_text(debates_body)
  #check if function generate the required length of results
  expect_equal(length(x[[1]]), 6)
  expect_equal(length(x[[2]]), 6)
  expect_equal(length(x[[3]]), 6)
  expect_equal(length(x[[1]][[6]]), 3)
  #check if output of the function is a list
  expect_type(x, "list")
})
```

Election Year	Candidates	Number of Chunks
2016	CLINTON	87
2016	TRUMP	124
2012	OBAMA	56
2012	ROMNEY	71
2008	OBAMA	63
2008	MCCAIN	63
2004	KERRY	33
2004	BUSH	41
2000	GORE	49
2000	BUSH	56
1996	CLINTON	45
1996	DOLE	46

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

```
generate_sent <- function(x) {
  if(is.null(x)){
    stop("Your list of speech lines is empty. Please provide another one.")
  }

  #initialize the list storing speech sentences
  speech_sent <- x
  #loop through each speech text chunk for each candidate
  for (i in 1:length(x)) {
    for (j in 1:length(x[[i]])) {
      for (m in 1:length(x[[i]][[j]])) {
        speech_sent[[i]][[j]][[m]] = as.list(speech_sent[[i]][[j]][[m]])
        #locate the endpoint of each sentence; In order to exclude URL, replace www. with www_
        link_rep = sapply(x[[i]][[j]][[m]], str_replace_all, pattern = 'www\\.', replacement = 'www_')
        loc = sapply(link_rep, str_locate_all,
                     pattern = '[[:upper:]]?[[:lower:]]{2,}[\\.+|?|!]')[[1]][, 2]

        #initialize a temp list to store the info
        tmp_list = vector('list', length(loc))

        #create substrings of text chunk for each sentence
        if (length(loc) > 0) {
          for (n in 1:length(loc)) {
            if (n == 1) {
              tmp_list[n] = sapply(x[[i]][[j]][[m]],
              substr,
              start = 1,
              stop = loc[n])
            }
            else {
              tmp_list[n] = sapply(x[[i]][[j]][[m]],
              substr,
              start = loc[n - 1] + 1,
              stop = loc[n])
            }
          }
          #remove the parts like "HILTON:" and leading and trailing blanks
          tmp_list = lapply(tmp_list, str_remove_all, pattern = '[[:upper:]]{4,12}:')
          speech_sent[[i]][[j]][[m]] = lapply(tmp_list, str_trim, side = c("both", "left", "right"))
        }
      }
    }
  }
  return(speech_sent)
}

speech_sent = generate_sent(speech_lines)

test_that("generate_sent works with normal input", {
  x = generate_sent(speech_lines)
  #check if function generate the required length of results
  expect_equal(length(x), 6)
  expect_equal(length(x[[6]][[3]]), length(speech_lines[[6]][[3]]))
  #check if output of the function is a list
  expect_type(x, "list")
})
```

```
generate_word <- function(x) {
  if(is.null(x)){
    stop("Your list of speech sentences is empty. Please provide another one.")
```

```r
  }
  cond =
    "[[:alpha:]]{1,}[,]?[[:space:]]+|[[:alnum:]]{1,}[\\.+|?|!|;]|[[:alpha:]]{1,}[']{[:alpha:]]{1,2}"
  speech_word <- x
  for (i in 1:length(x)) {
    for (j in 1:length(x[[i]])) {
      #for each candidate's sentence, extract the seperate words
      speech_word[[i]][[j]] = lapply(x[[i]][[j]], str_extract_all, pattern = cond)
      #remove the leading or trailing blanks, or punctuations in each word
      for (m in 1:length(speech_word[[i]][[j]])) {
        speech_word[[i]][[j]][[m]] = sapply(speech_word[[i]][[j]][[m]], str_remove_all,
                                            pattern = '[[[:space:]]|,|\\.+|?|!|;]')
      }
    }
  }
  return(speech_word)
}

speech_word = generate_word(speech_sent)

test_that("generate_word works with normal input", {
  x=generate_word(speech_sent)
  #check if function generate the required length of results
  expect_equal(length(x),6)
  expect_equal(length(x[[6]][[2]][[6]]),length(speech_sent[[6]][[2]][[6]]))
  #check if output of the function is a list
  expect_type(x,"list")
})
```

(c) 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 and make of the word length for the candidates. Comment briefly on the results.

```r
counting <- function(x) {
  if(is.null(x)){
    stop("Your list of speech words is empty. Please provide another one.")
  }

  word_count <- char_count <- rep(list(list()), length(x))
  for (i in 1:length(x)) {
    #initialize the word and char count for each candidate
    word_count[[i]] <- char_count[[i]] <- c(rep(0, length(x[[i]])))
    names(word_count[[i]]) <- names(char_count[[i]]) <- names(x[[i]])
    for (j in 1:length(x[[i]])) {
      for (m in 1:length(x[[i]][[j]])) {
        # for each sentence spoken by each candidate, get the number of words and sum them up
        word_count[[i]][[j]] = word_count[[i]][[j]] + sum(sapply(x[[i]][[j]][[m]], length))
        for (n in 1:length(x[[i]][[j]][[m]])) {
          # for for each word spoken by each candidate, get the number of chars and sum them up
          char_count[[i]][[j]] = char_count[[i]][[j]] + sum(sapply(x[[i]][[j]][[m]][[n]], nchar))
        }
      }
    }
  }
  result = list(word_count, char_count)
  return(result)
}

word_count = counting(speech_word)[[1]]
char_count = counting(speech_word)[[2]]

test_that("counting works with normal input", {
  x = counting(speech_word)
  #check if function generate the required length of results
  expect_equal(length(x[[1]]), 6)
  expect_equal(length(x[[1]][[1]]), 3)
  #check if output of the function is a list of doubles
  expect_type(x[[1]][[1]], "double")
})
```

Election Year	Candidates	Number of Words	Number of Characters	Ave Word Length
2016	CLINTON	6276	27015	4.304493
2016	TRUMP	8356	35108	4.201532
2012	OBAMA	7198	32178	4.470408
2012	ROMNEY	7676	33309	4.339370
2008	OBAMA	7547	33065	4.381211
2008	MCCAIN	7068	31268	4.423882
2004	KERRY	7049	30300	4.298482
2004	BUSH	6295	27205	4.321684
2000	GORE	7129	31089	4.360920
2000	BUSH	7377	31845	4.316795
1996	CLINTON	7597	33251	4.376859
1996	DOLE	7969	34461	4.324382

Answer: According to the results, there's no significant difference between the candidates' average word length. And there's no trend like candidate who speaks longer words has higher chance to win.

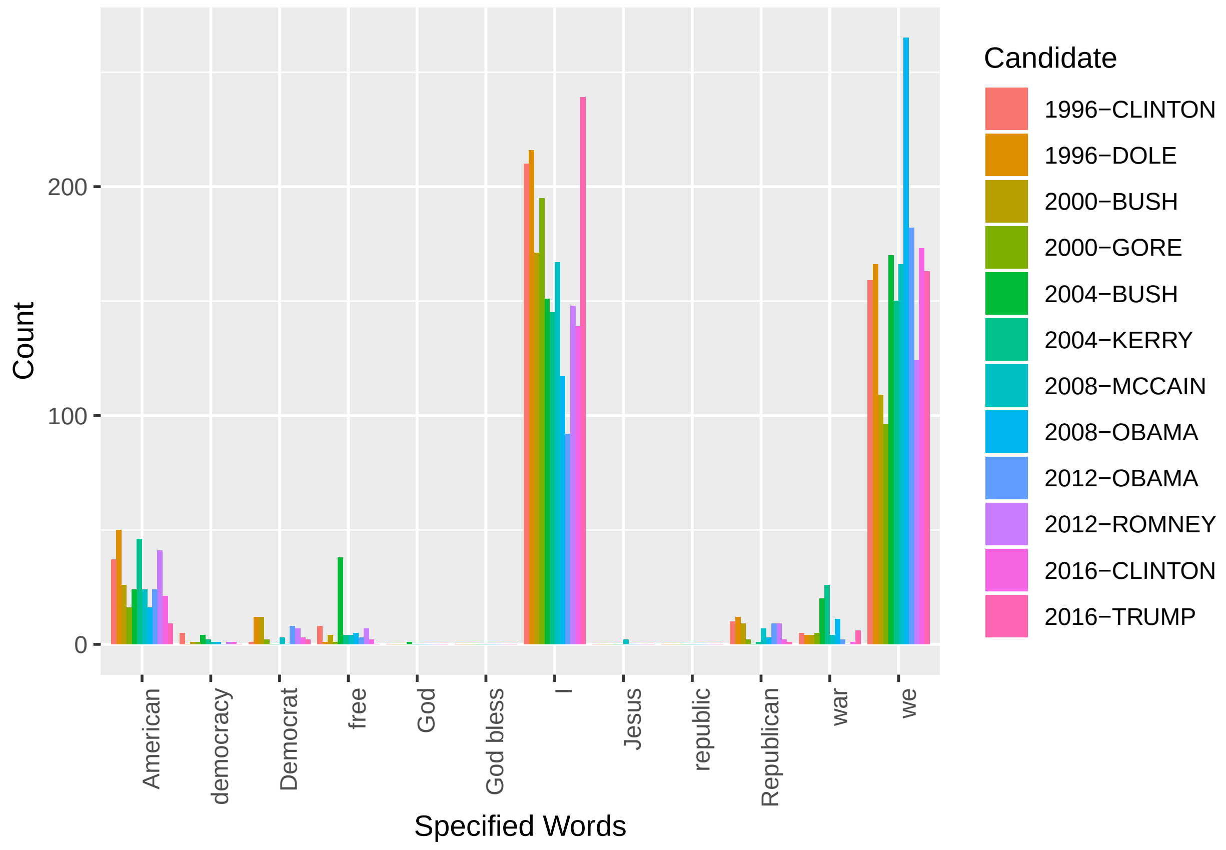
(d) For each candidate, count the following words or word stems and store in an R data structure: I, we, America{,n}, democra{cy,tic}, republic, Democrat{,ic}, Republican, free{,dom}, war, God [not including God bless], God Bless, {Jesus, Christ, Christian}. Make a plot or two and comment briefly on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

```r
words_ls = c('I', 'we', 'American', 'democracy', 'republic', 'Democrat', 'Republican',
          'free', 'war', 'God', 'God bless', 'Jesus')
#list of patterns used for regex processing
pattern_ls = c('[[:space:]]?I[[:space:]]', '[[:space:]]?[wW][eE][[[:space:]]|\']',
               'America[n]?', 'democracy|democratic', 'republic', 'Democrat[ic]?',
               'Republican', '[Ff]ree[dom]?', '[Ww]ar[[:space:]]',
               'God[[:space:]][^[Bb]less]', 'God\t[Bb]less', 'Jesus|Christ|Christian')
```

```r
wordstem_ct <- function(x, words, pattern) {
  if(is.null(x)){
    stop("Your list of speech lines is empty. Please provide another one.")
  }
  #initialize the list storing the word count
  word_stem = rep(list(list()), length(x))
  for (i in 1:length(x)) {
    #initialize the list for each candidate
    word_stem[[i]] = rep(list(rep(0, length(pattern_ls))), 2)
    names(word_stem[[i]]) = names(x[[i]])[2:3]
    for (j in 2:3) {
      names(word_stem[[i]][[j - 1]]) = words
      for (m in 1:length(pattern_ls)) {
        #for each speech text chunk for each candidate, count number of specified word and sum them up
        word_stem[[i]][[j - 1]][m] = sum(unlist(lapply(x[[i]][[j]], str_count, pattern = pattern[m])))
      }
    }
  }
  return(word_stem)
}

word_stem = wordstem_ct(speech_lines, words_ls, pattern_ls)

test_that("wordstem_ct works with normal input", {
  x = wordstem_ct(speech_lines, words_ls, pattern_ls)
  #check if function generate the required length of results
  expect_equal(length(x),6)
  expect_equal(length(x[[1]][[1]]),12)
  #check if output of the function is a list of doubles
  expect_type(x[[6]][[2]][[12]],"double")
})
```



According to the graph, "we" and "I" are the words candidates use most frequently. And the highest winner belongs to Obama with word "We" in 2008 election year. Candidates barely use word like God, Jesus, Christ or Christian. Maybe because America is a country of immigrants, it has diversity in religion beliefs. Only mentioning one religion may exert negative effects on election.

(e) Please include unit tests for your various functions. For the sake of time, you can keep this to a test or two for each function.

Problem 3

This problem asks you to design an object-oriented programming (OOP) approach to the debate text analysis of problem 2. You don't need to code anything up, but rather to decide what the fields and methods you would create for a class that represents a debate, with additional class(es) as needed to represent the spoken chunks and metadata on the chunks for the candidates. The methods should include methods that take input text and create the fields in the classes. You can think of this in the context of R6 classes, or in the context of classes in an object-oriented language like Python or C++. To be clear, you do not have to write any of the code for the methods nor even formal code for the class structure; the idea is to design the formats of the classes. Basically if you were to redesign your functional programming code from problem 2 to use OOP, how would you design it? As your reponse for each class, please provide a bulleted list of methods and bulleted list of fields and for each item briefly comment what the purpose is. Also note how each method uses other fields or methods.

Answer:

1. (Parent Class) class debate:

- Field: (1) debates content: which store the contents of years of debates; (2) debates year: which represents the year when debates occured;
- Method: (1) getContent()/setContent(): return or set the content of debates; (2) getYear()/setYear(): get or set the election years based on the debates content;

2. (Sub Class) class candidate_debate:

- Field: (1) everything inherits from parent class debate; (2) candidate names: list of candidates partipate in debates; (3) candidate contents: debates content for each candidate;
- Method: (1) getCandidate()/setCandidate(): get or set the candidate names for each speech chunk; (2) getCandCont()/setCandCont(): get or set the speech chunks for candidates;

3. (Sub Class) class candidate_sentence:

- Field: (1) everything inherits from class candidate_debate; (2) candidate sentences: speech sentences for each candidate;
- Method: (1) getSent/setSent(): get or set the speech sentences for candidates;

4. (Sub Class) class candidate_word:

- Field: (1) everything inherits from class candidate_sentence; (2) candidate words: speech words for each candidate;
- Method: (1) getWord/setWord(): get or set the speech words for candidates;