STA141 Assignment 6

Zhen Zhang

December 8, 2015

I did this assignment by myself and developed and wrote the code for each part by myself, drawing only from class, section, Piazza posts and the Web. I did not use code from a fellow student or a tutor or any other individual.

Part 1

algorithm

First I will explain the algorithm I am doing here.

Since I need to scrape all the posts or the posts needed by the user, the top level should be what the total posts. And this is what the function `post_summary` is doing. But each page should be scraped one by one, so I write another function `post_summary_page` to do this job.

And now we have come to the page level. By default, after one page is scraped, we need the url of the next page. So `get_next_page` accepts the tree (which is the result of the htmlParse result) as input, and get the next page's url by looking at 'next' tag and get the relativeURL.

Now what needs to be done is the main part, which is scraping all the posts within one page. I write a function called `post_summary_each` to do this, and it will call some small functions to scrape all the information, including author, reputation and so on.

The small pieces of functions, such as `scrape_reputation`, `scrape_view`, are extracting the information needed from the posts. The main function is `xpathSApply`. This is at the level of each post. After I get the result, combine them into a list. The reason I am not combining them as a dataframe now is because when there is no author information, for example, xpathSApply cannot find anything, then the value will be empty. If afterwards I combine the 15 small dataframes together into a dataframe, the empty cell will be occupied by its next value, causing the length to be different. So I modify the empty value to NA in the list, and then transform back to a dataframe. This is what `list_to_dataframe` is doing here. Then I am free to combine 15 dataframes to form a medium dataframe of each page, and combining the dataframes at page level together to get the dataframes at the top level.

Note the function `post_summary` accepts three parameters: tag, number of pages and number of posts. If number of pages is given, then the function will scrape the pages needed by the user. If the number of posts is given, then the function will first transform the number of posts to number of pages along with a reminder, and only select the first some pages indicated by the reminder in the last number of pages. If neither of them is given, it will scrape all the posts down to local.

Now let's see the functions. I will also give examples to test my function is working well.

functions

`list_to_dataframe`:

When we are scraping the posts from the stackoverflow, there will be some cases that the designed xpath pattern cannot find the right information. There are many reasons that can lead to this phenomenon, one is the information has been moved into other tags, and the other is this post indeed does not have this information. The first circumstance can be solved when we are scraping, and the second can be done by setting the empty information to NA. This function is designed to achieve this goal.

```
library(XML)

list_to_dataframe <- function(data) {
  # This function accepts a list as the input, then set empty result to NA, and transform
  # it to the dataframe

  # set no result to NA
  data <- lapply(data, function(x) {
    if (length(x) == 0) {
      x <- NA
    }
    x
  })

  # return the dataframe, also set stringsAsFactors option to FALSE
  result <- as.data.frame(data, stringsAsFactors = FALSE)
  # remove rowname
  rownames(result) <- NULL

  result
}
```

small pieces of functions getting the information:

`scrape_author`:

Most of the time, I can select the information by using `".//div[@class = 'user-details']/a/@href[contains(., 'users')]/.."`, here I use contains, because in community wiki, stackoverflow will track the revision history and then get multiple results when extracting user. And the user we want is the user with its name, so the href with `/user/userid/name` is our interest. But there are some circumstances that this cannot achieve the goal, for example, no value for the a tag. Then we cannot use the above method to extract author information. In that case, there is no `a` tag. so I extract it directly from the user-details class

```
# Next I will present some little functions that can scrape the corresponding data from
# the post summary

scrape_author <- function(tree) {
  # get author, accept a tree as input, and get the result as a vector of authors

  # situation I
  author <- xpathSApply(tree, ".//div[@class = 'user-details']/a/@href[contains(., 'users')]/..",
                        xmlValue, trim = TRUE)
  # situation II
  if (length(author) == 0) {
    author <- xpathSApply(tree, ".//div[@class = 'user-details']", xmlValue, trim = TRUE)
  }

  author
}
```

`scrape_time`:

All the time, almost, the information about time lies in `.//div[@class = 'user-action-time']/span/@title`. This is the absolute time, and since it is string format, I use `as.POSIXct` to tranform it to R `POSIXct` format.

```
scrape_time <- function(tree) {
  # get time
  time <- xpathSApply(tree, ".//div[@class = 'user-action-time']/span/@title")
  # convert time to r standard POSIXct format
  time <- as.POSIXct(strptime(time, "%Y-%m-%d %H:%M:%S"))

  time
}
```

scrape_title:

.//div[@class = 'summary']/h3/a is enough here

```
scrape_title <- function(tree) {
  # get title
  title <- xpathSApply(tree, ".//div[@class = 'summary']/h3/a", xmlValue)

  title
}
```

scrape_reputation:

First, I will get the reputation using .//div[@class = 'user-details']/span[@class = 'reputation-score'], it will get most of the value. The result can contains ',', when the reputation is bigger than 1000. So use regular expression to remove ','. Another circumstance is, when the reputation is very large, using the previous method will give us, for example, 5k. Then I can use .//div[@class = 'user-details']/span[@class = 'reputation-score']/@title to extract the relevant information, which is the accurate information.

```
scrape_reputation <- function(tree) {
  # get reputation
  # situation I
  reputation <- xpathSApply(tree, ".//div[@class = 'user-details']
                            //span[@class = 'reputation-score']", xmlValue)
  reputation <- gsub('(,)', '', reputation)
  reputation <- as.numeric(reputation)
  # situation II
  if (length(reputation) == 0||is.na(reputation)) {
    reputation_revised_raw <- xpathSApply(tree, ".//div[@class = 'user-details']
                                          //span[@class = 'reputation-score']/@title")
    reputation <- gsub("[^0-9]*([0-9]*)", '\\1', reputation_revised_raw)
  }

  reputation
}
```

scrape_view:

In deed, we can extract value using this xpath syntax: .//div[@class = 'views]. But when the number of view is very large, say 4k views (with class views hot), we cannot get the accurate number. So a general method for this part is to extract the number from the title: .//div[@class = 'views ' or 'views hot']/@title

```
scrape_view <- function(tree) {
  # get scrape
  n_view_raw <- xpathSApply(tree, ".//div[@class = 'views ' or 'views hot']/@title")
```

```
  # get the view digit using regular expression
  n_view <- unlist(strsplit(n_view_raw, ' '))[1]
  n_view <- gsub(',', '', n_view)
  n_view <- as.numeric(n_view)

  n_view
}
```

scrape_answer:

.//div[@class = 'stats']/div[2]/strong is enough here.

```
scrape_answer <- function(tree) {
  # get the number of answers
  n_ans <- xpathSApply(tree, ".//div[@class = 'stats']/div[2]/strong", xmlValue)
  n_ans <- as.numeric(n_ans)

  n_ans
}
```

scrape_vote:

.//span[@class = 'vote-count-post ']/strong is enough here

```
scrape_vote <- function(tree) {
  # get the number of votes
  n_vote <- xpathSApply(tree, ".//span[@class = 'vote-count-post ']/strong", xmlValue)
  n_vote <- as.numeric(n_vote)

  n_vote
}
```

scrape_url:

.//div[@class = 'summary']/h3/a/@href is enough here

```
scrape_url <- function(tree) {
  # get the url
  url <- xpathSApply(tree, ".//div[@class = 'summary']/h3/a/@href")

  url
}
```

scrape_id:

use ./@id to get the id, and we will get result like "question-summary-34171330". Then use regular expression to extract the digit

```
scrape_id <- function(tree) {
  # get the raw id
  id_raw <- xpathSApply(tree, "./@id")
  # use regular expression to parse the id
  id <- gsub('.*-([0-9]+)', '\\1', id_raw)
  id <- as.numeric(id)
```

```
  id
}
```

scrape_tag:

use `.//div[@class = 'summary']/div/@class[contains(., 'tag')]/..`, because I can extract tag as result of "r python ruby". Here I use contain, since there are many divs at that level, and the div contains tag is what I need. The result I get will be all the divs that are tags, namely "r python ruby", combined in one string

```
scrape_tag <- function(tree) {

  tags_raw <- xpathSApply(tree, ".//div[@class = 'summary']/div/@class[contains(., 'tag')]/..",
                          xmlValue, trim = TRUE)
  # substitute space with space and ;
  tags <- gsub('( )', '; ', tags_raw)

  tags
}
```

function at post level

post_summary_each is the function that do the parsing for each post. The tree here is at the post level. It calls many small functions for each column. At last, first combine them into a list, and after empty is transformed to NA, the list is transformed back to a dataframe as a final output

```
post_summary_each <- function(tree) {

  author <- scrape_author(tree)
  time <- scrape_time(tree)
  title <- scrape_title(tree)
  reputation <- scrape_reputation(tree)
  n_view <- scrape_view(tree)
  n_ans <- scrape_answer(tree)
  n_vote <- scrape_vote(tree)
  url <- scrape_url(tree)
  id <- scrape_id(tree)
  tags <- scrape_tag(tree)

  # combine the results above into a list
  data <- list(id = id, date = time, tags = tags, title = title, url = url,
               views = n_view, votes = n_vote, answers = n_ans, user = author,
               reputation = reputation)
  # call list_to_dataframe to transform it into a dataframe
  result <- list_to_dataframe(data)

  # return
  result
}
```

function at page level:

post_summary_page is the function that parse the page of each tag. It will accept the tree and split it into sub trees, and render each tree to post_summary_each, then combine all the data got from that function. It also can only subset a small subset of the tree, when the n_post argument is given by the user

```
post_summary_page <- function(tree, n_post = 0) {

  # split into sub trees
  tree <- xpathSApply(tree, "//div[@class = 'question-summary']")
  # If n_posts is pointed by the user, only subset part of it
  if (n_post != 0) tree <- tree[1:n_post]

  # for each sub tree, call post_summary_each, and get the result as a list
  data_list <- lapply(tree, function(x) {
    post_summary_each(x)
  })
  # use do.call to combine into a big dataframe
  data <- do.call(rbind, data_list)

  # return
  data
}
```

function to get the next page:

get_next_page is served as finding the next url page. It knows the current tree and the initial url, then extract the next page information from the tree and combine with the initial url to find the next url by the function getRelativeURL. If there is no next page, this function will return NA, then the interface knows it is time to stop finding the next page

```
get_next_page <- function(tree, init_url) {

  # initialize the result
  next_url <- NA
  # extract the information of the next page url
  next_page_href <- xpathSApply(tree, "//a[@rel = 'next']/@href")
  # get next url by the function getRelativeURL, if there is next page, otherwise
  # return initial value, which is NA.
  if (!is.null(next_page_href)) {
    next_url <- getRelativeURL(next_page_href, init_url)
  }

  # return url
  next_url
}
```

Interface function to the user:

post_summary is the function interface provided to the user with three parameters, tag, n_pages and n_posts. The first parameter, tag, points to which category of questions we are interested in. It can be r, python, c and so on. The second parameter, n_pages, is the number of pages the user wants. If the user provides a value, for example, 10, it will extract the first 10 pages data, in default a dataframe with 150 rows. The third parameter n_posts, is the number of posts, and the function will extract the corresponding number from the tag. But if the user does not provide n_pages or n_posts argument, then this function will scrape all the

data from this category. At last, all the dataframe will be combined together and return a final dataframe as output

```
post_summary <- function(tag, n_pages = 0, n_posts = 0) {

  # transform the number of posts to the number of pages, with the reminder as n_reminder
  if (n_posts != 0) {
    n_pages <- ceiling(n_posts / 15)
    n_reminder <- n_posts - (n_pages - 1) * 15
  } else {
    n_reminder <- 15
  }

  # This is the initial url from the tag
  init_url <- sprintf("http://stackoverflow.com/questions/tagged/%s", tag)
  # the tree of the first page
  tree <- htmlParse(init_url)
  # the first page's data
  data_new <- post_summary_page(tree)
  # give the value to data
  data <- data_new

  # This is a while loop takes one condition. The condition is taken into account
  # when the number of pages is limited by the user. If the page range is not limited,
  # The function will scrape all the data under the tag, until no more post. This
  # is achieved by the break statement below, given by next_url
  while (n_pages != 1) {
    # subtract the number of pages by one, to get the number of pages still need to
    # be extracted
    n_pages <- n_pages - 1

    # get the next page url
    next_url <- get_next_page(tree, init_url)
    # test if there is next page's url, if not, jump out of the while loop
    if (is.na(next_url)) break
    # get the next page's tree
    tree <- htmlParse(next_url)

    # if there is only one page left now, use the number of reminder as the last n posts
    # to get the new page's data
    if (n_pages == 1) {
      data_new <- post_summary_page(tree, n_reminder)
    } else {
      data_new <- post_summary_page(tree)
    }

    # combine the new data with the old data
    data <- rbind(data, data_new)
  }

  # return data
  data
}
```

Test some result

Here I will select some data for test (use tbl_df in dplyr package to shown the result briefly):

The n_pages is given:

```
r_post_data <- post_summary('r', n_pages = 2)
# show part of the results I get
tbl_df(r_post_data)
```

```
## Source: local data frame [30 x 10]
##
##          id                date                              tags
##       (dbl)              (time)                             (chr)
## 1  34188472 2015-12-09 20:34:59    r; plot; ggplot2; time-series
## 2  34188128 2015-12-09 20:12:45               r; hadoop; rhadoop
## 3  34188023 2015-12-09 20:06:31                    r; quantmod
## 4  34187995 2015-12-09 20:05:15 r; time-series; cross-validation
## 5  34187909 2015-12-09 19:59:16                     r; ggplot2
## 6  34187861 2015-12-09 19:56:35                        r; rook
## 7  34187831 2015-12-09 19:54:46                      r; matlab
## 8  34187742 2015-12-09 19:50:16                     r; ggplot2
## 9  34187722 2015-12-09 19:49:11       r; parsing; csv; condition
## 10 34187641 2015-12-09 19:44:04                               r
## ..      ...                 ...                             ...
## Variables not shown: title (chr), url (chr), views (dbl), votes (dbl),
##   answers (dbl), user (chr), reputation (dbl)
```

The n_posts is given:

```
r_post_data <- post_summary('r', n_posts = 16)
# show part of the results I get
tbl_df(r_post_data)
```

```
## Source: local data frame [16 x 10]
##
##          id                date                                      tags
##       (dbl)              (time)                                     (chr)
## 1  34188472 2015-12-09 20:34:59             r; plot; ggplot2; time-series
## 2  34188128 2015-12-09 20:12:45                        r; hadoop; rhadoop
## 3  34188023 2015-12-09 20:06:31                             r; quantmod
## 4  34187995 2015-12-09 20:05:15          r; time-series; cross-validation
## 5  34187909 2015-12-09 19:59:16                              r; ggplot2
## 6  34187861 2015-12-09 19:56:35                                 r; rook
## 7  34187831 2015-12-09 19:54:46                               r; matlab
## 8  34187742 2015-12-09 19:50:16                              r; ggplot2
## 9  34187722 2015-12-09 19:49:11                r; parsing; csv; condition
## 10 34187641 2015-12-09 19:44:04                                        r
## 11 34187579 2015-12-09 19:40:14                          python; r; rpy2
## 12 34187487 2015-12-09 19:33:40 r; vector; scripting; element; multiplying
## 13 34187451 2015-12-09 19:31:51                       c; r; class; oop
## 14 34187339 2015-12-09 19:26:09                                        r
## 15 34187333 2015-12-09 19:25:38                             r; dplyr
```

```
## 16 34187162 2015-12-09 19:16:50                                        r; shiny
## Variables not shown: title (chr), url (chr), views (dbl), votes (dbl),
##   answers (dbl), user (chr), reputation (dbl)
```

I will scrape all data from the r tag. I will not run it here, since it consumes too much time

```
# remove duplicate posts
r_post_data <- unique(r_post_data)
```

Part 2

Algorithm

First I would like to talk about the algorithm I followed in this part. First I use `htmlParse` to get the whole tree, and then split them into blocks. Here I define the block as a combination of question or answer with its combined comments. So now I come from tree level to block level.

For a block, I identify that it consists of question/answer with comments. They lie in two tables (Although there is no comment with this q/a, there is still such table for comment, but no contents). Then I split the two tables, the first one is given to `parse_qa`, and the second one is given to `parse_comment`.

The `parse_qa` can be used to parse question as well as answer, since they share the same pattern, and comment has a different pattern, so a different function `parse_comment` is written. These two functions will give a dataframe for the relevant information, and then after combining these sub dataframes together we can have the final dataframe.

Functions:

block level:

`split_block`: This function takes a htmlParse Tree as input and then split it into blocks. After the blocks have been analyzed, combine the dataframes together

```
split_block <- function(tree) {

  # split the tree into blocks
  tree_split <- xpathSApply(tree, "//div[@id = 'mainbar']//div[@class = 'question'
                            or @class = 'answer' or @class = 'answer accepted-answer']")
  # render different blocks to the function parse_block
  data_list <- lapply(tree_split, parse_block)
  # combine the dataframes together
  data <- do.call(rbind, data_list)

  # Different dataframes has the same qid within one questions
  data$qid <- data$id[1]

  data
}
```

`parse_block` is used to parse blocks. Since a block consists of two tables, and the first is for q/a, the second is for comment. The q/a part is rendered to `parse_qa`, the comment part is rendered to `parse_comment`. And since comments may be null, so when combining these two parts together, I need to consider this circumstance, so a test command is written to handle this situation

```
parse_block <- function(block) {

  # extract type from the class attribute. I use the class attribute since it is consistent
  # among question, answer and answer accepted-answer. There three attribute are what I will
  # get here
  type <- xpathSApply(block, './@class')
  # use regular expression to extract the first word, either question or answer
  type <- unlist(strsplit(type, ' '))[1]
  # get the pid, since the parent id is the same within one block
  pid <- xpathSApply(block, sprintf('./@data-%sid', type))

  # split the block into two tables
  tables <- xpathSApply(block, './table/tr')
  # pass table one to parse_qa
  data_part_one <- parse_qa(tables[[1]], pid, type)
  # pass table two to parse_comment
  data_part_two <- parse_comment(tables[[2]], pid, type)

  # give the q/a data to result
  result <- data_part_one
  # test if comment data is null. If not, combine it with the result above
  if (length(data_part_two) != 0) {
    result <- rbind(result, data_part_two)
  }

  # return
  result
}
```

Parse level:

parse_qa: This function is served as extract question and answer, since they share the same pattern. Also, for answers, there are two types: answers or accepted answers. At last, combine into a list, remove the null ones, convert to a dataframe

```
parse_qa <- function(tree, pid, type) {

  # extract user block
  user_block <- xpathSApply(tree, ".//td/@class[contains(., 'post-signature')]/..")
  # But usually there will be multiple authors within one block. The reason is, for one post,
  # no matter answer or question, there is usually two types of authors: the one editted it
  # and the one posted it. And the one posted it is our interest, which usually comes at the
  # last position
  user_block <- user_block[[length(user_block)]]

  # All the following parts related to author information depend on the user_block,
  # including user, userid, date, reputation

    # extract user info
    user_info <- xpathSApply(user_block, ".//div[@class = 'user-details']/a
                             /@href[contains(., 'users')]")
    # split by /
    user_info_split <- unlist(strsplit(user_info, '/'))
```

```
    # the name is the last element
    user <- user_info_split[length(user_info_split)]
    # the id is the last two element
    userid <- user_info_split[length(user_info_split) - 1]

    # extract date
    date <- xpathSApply(user_block, ".//span[@class = 'relativetime']/@title")

    # get reputation
    reputation <- xpathSApply(user_block, ".//span[@class = 'reputation-score']",
                              xmlValue)
    # sometimes there is a comma in reputation, like 5,990. So remove the comma:
    reputation <- gsub('(,)', '', reputation)
    reputation <- as.numeric(reputation)
    # but there are some reputations that are in the form of 5k, or no reputation data in
    # the above method. So to get the accurate number, I use the following xpath syntax:
    if (length(reputation) == 0||is.na(reputation)) {
      reputation_revised_raw <- xpathSApply(user_block, ".//span[@class = 'reputation-score']/
                                            @title")
      reputation <- gsub("[^0-9]*([0-9]*)", '\\1', reputation_revised_raw)
    }

  # get score
  score <- xpathSApply(tree, ".//span[@class = 'vote-count-post ']", xmlValue)
  # transform to numeric
  score <- as.numeric(score)

  # get content
  content <- xpathSApply(tree, ".//div[@class = 'post-text']", xmlValue)

  # get post id
  id <- as.numeric(pid)

  # construct dataframe
  data <- list(id = id, type = type, user = user, userid = userid, date = date,
               reputation = reputation, score = score, content = content,
               pid = pid, parent = NA)
  # transform list to dataframe
  data <- list_to_dataframe(data)

  # return value
  data
}
```

`parse_comment`: This function is served as parse comment from a tree. Since in some situations the comments will be empty, then I need to initialize an empty dataframe to avoid no return value error. After that, different details are extracted.

```
parse_comment <- function(tree, pid, parent) {

  # initialize result
  result <- data.frame()
```

```r
# get type
type <- 'comment'

# get user info, the same logic as that of question and answer
user_info <- xpathSApply(tree, ".//div[@class = 'comments ']
                         //div[@class = 'comment-body']/a/@href")

# This will only be done if the user_info is not empty
if (!is.null(user_info)) {
  # split
  user_info_split <- strsplit(user_info, '/')
  # extract author name
  user <- sapply(user_info_split, `[`, 4)
  # extract author id
  userid <- sapply(user_info_split, `[`, 3)

  # get the date
  date <- xpathSApply(tree, ".//div[@class = 'comments ']
                      //span[@class = 'relativetime-clean']/@title")

  # get reputation
  reputation_raw <- xpathSApply(tree, ".//div[@class = 'comments ']
                                //div[@class = 'comment-body']/a/@title")
  # get reputation value
  reputation <- gsub("([0-9]+).*", '\\1', reputation_raw)
  # transform to numeric
  reputation <- as.numeric(reputation)

  # get score
  score <- xpathSApply(tree, ".//div[@class = 'comments ']//td[@class = ' comment-score']",
                       xmlValue, trim = TRUE)
  # transform to numeric
  score <- as.numeric(score)

  # get content
  content <- xpathSApply(tree, ".//div[@class = 'comments ']//span[@class = 'comment-copy']",
                         xmlValue)

  # get id
  id_raw <- xpathSApply(tree, ".//div[@class = 'comments ']//tr[@class = 'comment ']/@id")
  # extract id
  id <- gsub("[^0-9]*([0-9]+)", '\\1', id_raw)

  # get the dataframe
  data <- data.frame(id = id, type = type, user = user, userid = userid, date = date,
                     reputation = reputation, score = score, content = content,
                     pid = pid, parent = parent)
  # transform list to dataframe
  result <- list_to_dataframe(data)
}

# return data
result
```

```
}
```

post-level function

scrape_post: This function is the connection function from url to final dataframe. First get the tree from the url, and the use split_block to get the final dataframe, and after that return the final result.

```
scrape_post <- function(url) {

  # get the tree
  tree <- htmlParse(url)

  # get the data
  data <- split_block(tree)

  # return data
  data
}
```

Test:

```
post_test <- scrape_post(
  "http://stackoverflow.com/questions/2564258/plot-two-graphs-in-same-plot-in-r?rq=1")
tbl_df(post_test)
```

```
## Source: local data frame [31 x 11]
##
##          id     type                user  userid                date
##       (chr)    (chr)               (chr)   (chr)               (chr)
## 1   2564258 question  sandra-schlichting  256439 2010-04-01 23:28:14Z
## 2  46337977  comment       isomorphismes  563329 2015-03-14 14:19:53Z
## 3   2564276   answer               bnaul  233293 2010-04-01 23:33:35Z
## 4  24471398  comment               frank   60628 2013-06-05 18:51:23Z
## 5  25509748  comment           soumendra  505493 2013-07-09 04:17:51Z
## 6  46338039  comment       isomorphismes  563329 2015-03-14 14:23:51Z
## 7  54303740  comment                kavi 4915000 2015-10-21 04:35:43Z
## 8  54339088  comment               bnaul  233293 2015-10-21 20:52:42Z
## 9   5046762   answer              402681   users                   NA
## 10  7650136  comment alessandro-jacopson   15485 2011-06-28 07:51:56Z
## ..      ...      ...                 ...     ...                 ...
## Variables not shown: reputation (chr), score (dbl), content (chr), pid
##   (chr), parent (chr), qid (chr)
```

Part 3

Load the data:

```
load("/Users/Elliot/Programming/R/WD/rQAs.rda")
load("/Users/Elliot/Programming/R/WD/r_post_data.RData")
```

Before all starts, I will first subset the rQAs to some subsets, since they will be used afterwards, and also transform date to POSIXct format. This transformation includes to questions, answers, answers and comments.

```
rQAs$date <- as.POSIXct(strptime(rQAs$date, "%Y-%m-%d %H:%M:%S"))
# get the unique dataframe
rQAs <- unique(rQAs)
rQAs$qid <- as.numeric(rQAs$qid)

rQAs_questions <- subset(rQAs, type == 'question')
rQAs_ans <- subset(rQAs, type == 'answer')
rQAs_ans_comment <- subset(rQAs, type %in% c('comment', 'answer'))
```

Here comes one of the most important part, linking the dataframe I got from Part 1 (r_post_data) with the dataframe from this Part (rQAs_questions). I will link the questions in rQAs with the data in r_post_data, and create a new table, rQAs_whole. Any comments or answers with respect to the question id should refer to the rQAs_whole dataframe

```
rQAs_whole <- inner_join(x = r_post_data, y = rQAs_questions[c('text', 'userid', 'qid')],
                         by = c('id' = 'qid'))
```

3.1

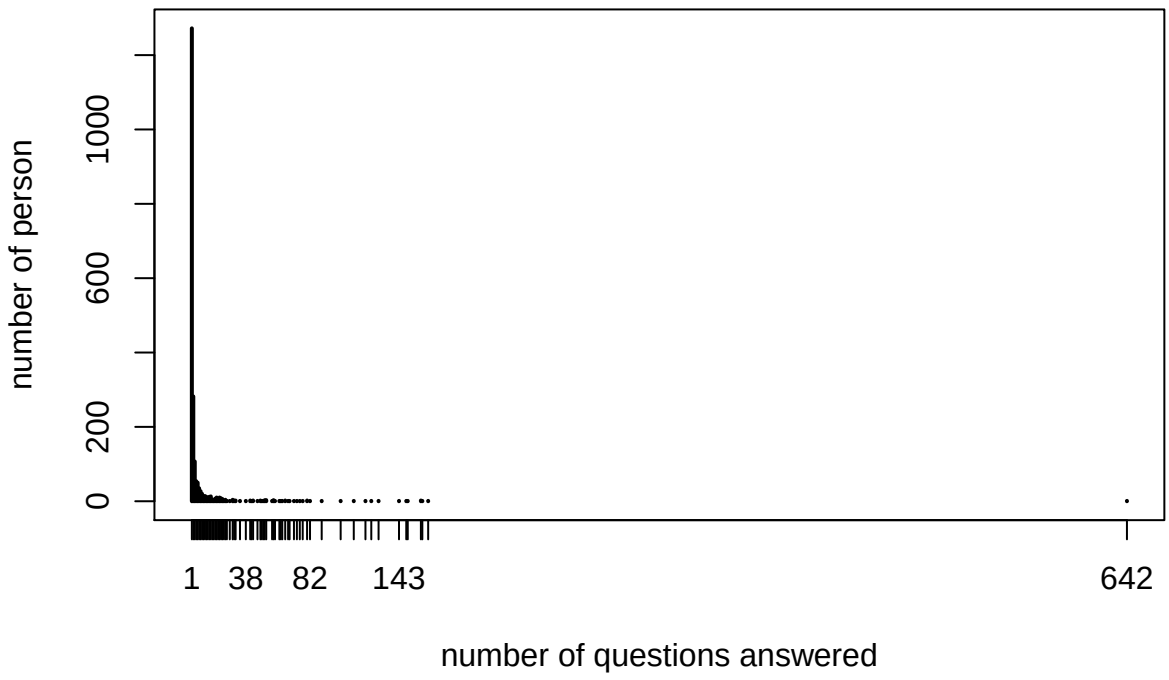
I will do two dataframes, from asnwer data and answer/comment data respectively.

The methodology I do here is first count the number of user splitted by user id, then plot the table value for different count group. Usually The smallest number is very much compared to big number, so I will restrict to a small interval to see the trend more clearly.

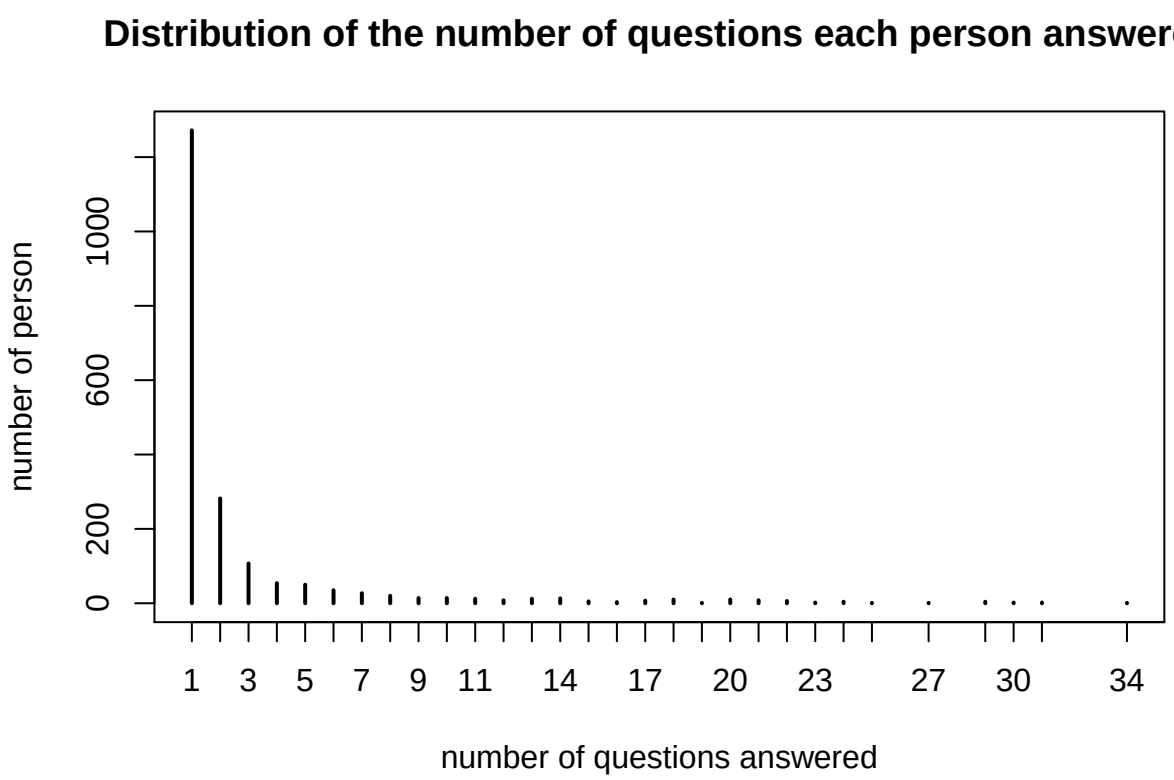
Answer data

```
user_count <- ddply(rQAs_ans, .(userid), function(x) data.frame(count = nrow(x)))

plot(table(user_count$count), xlab = 'number of questions answered',
     ylab = 'number of person',
     main = 'Distribution of the number of questions each person answered')
```




```
# It seems that most of the users are within 1-34 interval. So restrict the interval:
plot(table(user_count$count), xlim = c(1, 34), xlab = 'number of questions answered',
     ylab = 'number of person',
     main = 'Distribution of the number of questions each person answered')
```

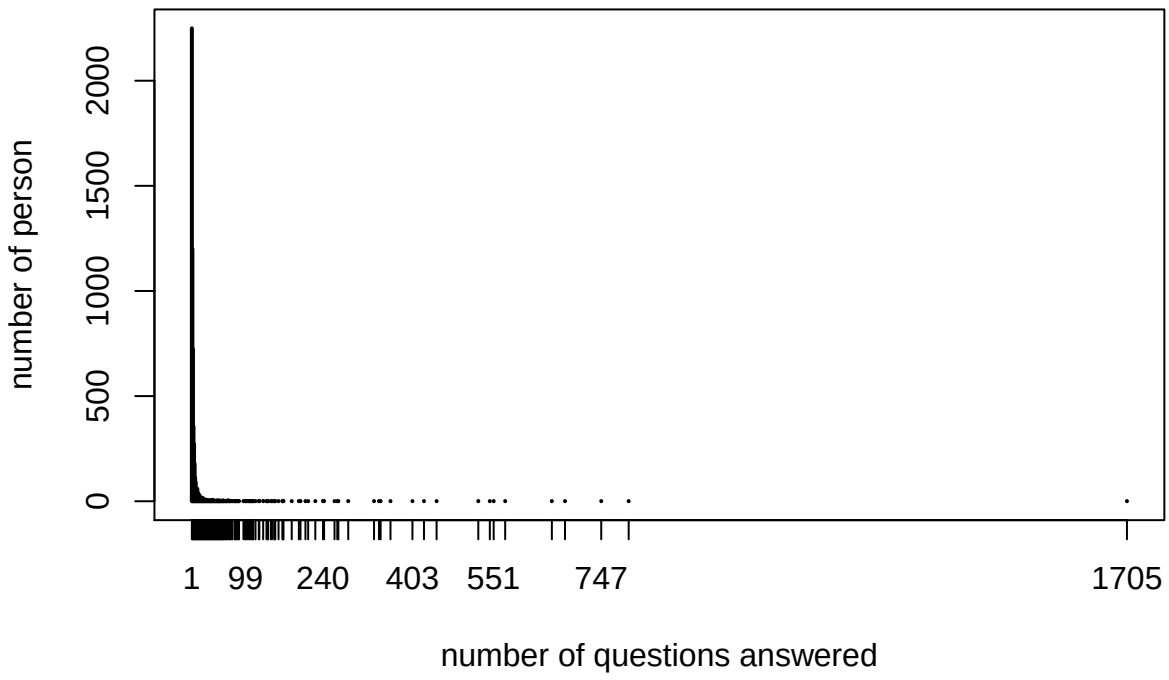


From the plot we know that most of the people in the r board answer less than 10 questions, and those answered 1-5 questions are the most.

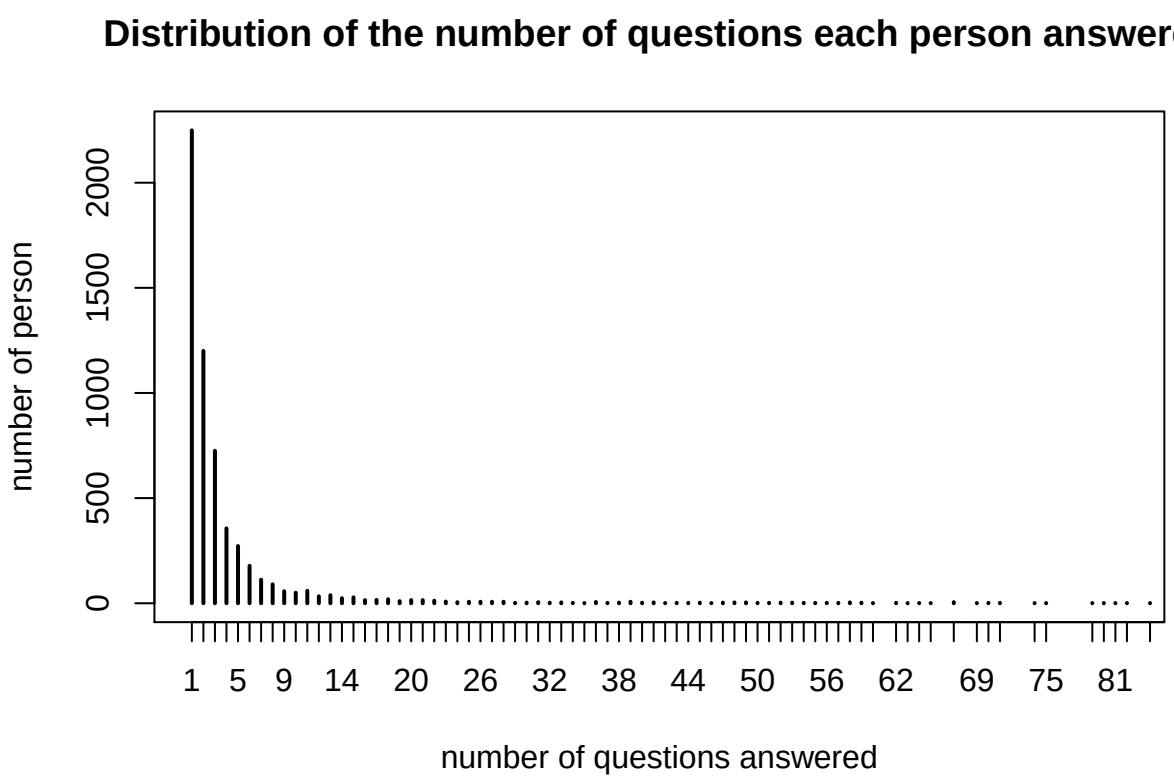
The answer and comment data

```
user_count <- ddply(rQAs_ans_comment, .(userid), function(x) data.frame(count = nrow(x)))

plot(table(user_count$count), xlab = 'number of questions answered',
     ylab = 'number of person',
     main = 'Distribution of the number of questions each person answered')
```




```
# It seems that most of the users are within 1-82 interval. So restrict the interval:
plot(table(user_count$count), xlim = c(1, 82), xlab = 'number of questions answered',
     ylab = 'number of person',
     main = 'Distribution of the number of questions each person answered')
```



From the plot we know that most of the people in the r board answer less than 20 questions, and those answered 1-5 questions are the most.

Above all we know that people tend to answer very small number, and only a very samll proportion of people answer many questions.

3.2

Before this question starts, I will first define a function that does the job of get the top frequency ones in a vector or a list. Since many of subsequent questions will use this function, it is time to write a function for it be elegant.

```
get_top <- function(input, n = 0, decreasing = FALSE) {
  # it will first transform the input into a vector, then table it, and at last find
  # the last/top n elements defined by the user. If no top value is given, it will return
  # the whole table in a increasing order. The top or last behavior is also defined by
  # the user, while if decreasing is FALSE, the last n elements will be returned, else
  # the top n will be returned

  # transform to vector
  result <- unlist(input)
  # table it
  result_table <- table(result)
  # get the result before get the last/top n elements
  result <- result_table[order(result_table, decreasing = decreasing)]
  # get the last/top n
```

```
  if (n != 0) {
    result <- result[1:n]
  }

  # return value
  result
}
```

I will get the tag directly from the data frame. But since the tags are separated by ';', I will split them, and then count the frequency of each tag

```
r_post_data_rQAs_tags_split <- strsplit(rQAs_whole$tags, '; ')
r_post_data_rQAs_tags_table_order <- get_top(r_post_data_rQAs_tags_split, decreasing = TRUE)

# show the top 20
r_post_data_rQAs_tags_table_order[1:20]
```

```
## result
##          r    ggplot2      shiny       plot data.frame data.table
##       8989        734        413        410        388        316
##      dplyr     matrix    rstudio      regex      knitr      loops
##        305        211        193        167        145        143
##   for-loop  rmarkdown   function        csv       list statistics
##        134        131        129        124        122        120
##     string     subset
##        110        103
```

The result shows that, excluding r tag, which will exist in all post since my data is scraped from r tag, the most tags are: ggplot2, shiny, plot, data.frame, data.table, dplyr, matrix and so on. This can remind us what topic, function or package is the most popular in the R community.

3.3

For this question, I understand 'about' this way: If the word 'ggplot' emerges in the text of the question, then I think the question is about ggplot. So `has_ggplot` can be evaluated as TRUE if a string has the word 'ggplot', otherwise FALSE.

```
has_ggplot <- function(text) {
  # parse ggplot
  result_ggplot <- grepl('ggplot', text, ignore.case = TRUE)

  result_ggplot
}
```

Now let's count how many of them:

```
sum(has_ggplot(rQAs_whole$text))
```

```
## [1] 914
```

Then there are 914 posts about ggplot. Considering there are 734 questions have ggplot2 as the tag, this result is reasonable.

3.4

In this question, I will find the word xml, html, web scraping are in the text of the question/answer/comment, and then group the result by qid. It means if one post in one pid has one of these three words, the result will be TRUE for that qid.

```
has_xml_html_scraping <- function(text) {
  # parse
  result <- grepl('xml|html|web scraping', text, ignore.case = TRUE)

  result
}
```

For each pid, merge all the text within that qid together. Then it is easily to find whether this pid has one of these three words

```
rQAs_pid_text_combination <- ddply(rQAs, .(qid), function(x) {
  paste0(x$text, collapse = '')
})
```

Now let's count how many of them:

```
sum(has_xml_html_scraping(rQAs_pid_text_combination$V1))
```

```
## [1] 1327
```

Then there are 1327 questions involve xml, html or web scraping

3.5

The function `parse_title` is the main function for this question. Having all the titles in hand, I will first split the title into words. Then for each of the words, I will test whether they are actually function based on the function is.function. If it is a function, the output of `is.function` will return TRUE, else FALSE.

But there is a situation I need to handle: is.function can only accept input of raw name of function. For example, `is.function(list)` will return TRUE while `is.function("list")` will return FALSE. And the output from title splitted results are all string, so I cannot pass the string to is.function directly. What I am doing here is to use the function get to get the function format of a string, for example, `get('ggplot')` will get the function ggplot and `is.function(get('ggplot'))` will be TRUE. When the object does not exist, which means the string is not a function, use tryCatch to catch the error, because if the function is not found, the expression will return error. So to avoid error, I use `tryCatch`, and set the error information to FALSE, as compared to TRUE, so afterwards I can select it

```
parse_title <- function(title) {
  # split the title
  title_split <- unlist(strsplit(title, ' '))

  # use is.function to test whether it is a function or not
  title_prase_function <- lapply(title_split, function(x) {
    tryCatch(is.function(get(x)), error = function(e) e <- FALSE)
  })
```

```
  # subset the title that is actually a function
  title_split[unlist(title_prase_function)]
}
```

```
# get the list that includes all the functions in each question title
title_function_list <- lapply(rQAs_whole$title, parse_title)
```

One more thing to note, by using this approach, I need to load the needed package into the r global environment. For instance, if ggplot is not loaded, then `is.function(ggplot)` will be FALSE. So I need to load some packages first. So what package should I load?

Remind 3.2, I have found the tags, so by observing the top tags, we know which tag is most popular, so I load some of the packages into the console.

```
r_post_data_rQAs_tags_table_order[1:20]
```

```
## result
##          r    ggplot2      shiny       plot data.frame data.table
##       8989        734        413        410        388        316
##      dplyr     matrix    rstudio      regex      knitr      loops
##        305        211        193        167        145        143
##   for-loop  rmarkdown   function        csv       list statistics
##        134        131        129        124        122        120
##     string     subset
##        110        103
```

So load ggplot2, shiny, dplyr, data.table

```
# table the list and order it
title_function_order <- get_top(title_function_list, decreasing = TRUE)
# get the top 20 functions
title_function_order[1:20]
```

```
## result
##     with     data      for function        -       by     plot        I
##     1311      999      793      500      443      400      360      350
##    frame       is     list     file      row   matrix       do   vector
##      331      289      274      262      238      235      218      218
##       as    table     time     each
##      201      184      182      157
```

Comment: But this is not accurate at all. As we can see, with and data are indeed functions, but I think what the author meant was not the function. It only served as the proposition.

Other strategy in this question: I can use `ls(package::'base')` to get the function in base. Also can get other functions in other packages. Then I get a list of functions. Then I can directly check whether the string in title match words in the list of functions. If so, it is a function, if not, it should not be a function.

3.6

In this question, I will get the comment and answer data, then extract code chunk from the data first.

```
extract_code <- function(text) {
  # extract code chunk, by the html tag <code></code>

  regmatches(text, gregexpr("<code>[^<]*</code>", text))
}
```

After that, I will use `parse_code` to find function inside code chunk. Here I find function are usually written like fun(, so I will grep it from the code chunk, and then remove left bracket (. I do so because I know there are some cases that functions are nested within each other, and also I don't want the arguments, and by only extracting (I can avoid these situations. Then I will remove (by gsub.

```
parse_code <- function(text) {

  result <- unlist(regmatches(text, gregexpr('[.a-zA-Z]+\\(', text)))
  result <- gsub('\\(', '', result)
  result
}
```

```
code_chunk <- extract_code(rQAs_ans_comment$text)
# parse the code function by parse_code
code_function <- unlist(lapply(code_chunk, parse_code))

code_function_order <- get_top(code_function, decreasing = TRUE)
# show the top 20
code_function_order[1:20]
```

```
## result
##          c    library   function       list data.frame        aes
##       9551       4442       3879       1986       1983       1639
##     length     lapply      names        rep     ggplot      paste
##       1604       1438       1139       1103       1021       1003
##        sum       nrow        seq         by     sapply as.numeric
##       1001        921        857        831        802        801
##      setDT      is.na
##        736        728
```

Here we can see the result is much more accurate now. The function c, library, list and dataframe are actually function we use frequently, so this method proves to be a good estimate. Also compared with 3.5, it is much better.