

Post 2: Text Mining and Word Clouds

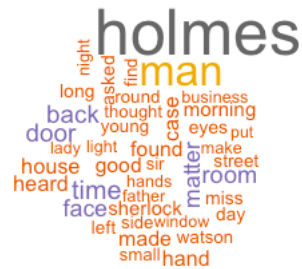
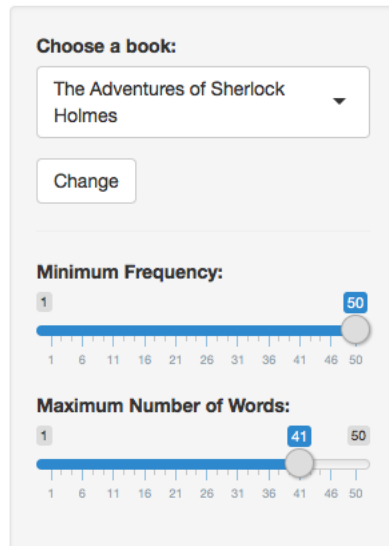
Timothy Tan

November 29, 2017

Introduction

Ever wondered how to get those fancy word clouds shown on the internet? Well, using some libraries and shiny app in R can allow you to create your very own! This post will walk you through the creation of a shiny application that will allow you to read some blocks of texts (in this post we will use some books), and create your own word cloud like the one below!

Word Cloud



Word Cloud!

Setting Up global.R

To get started, you will need to install and load the following packages: tm, wordcloud, and memoise, into a global.R file.

In shiny, using a global.R file will enable you to create objects and functions visible to both the server and the ui.

```
library(tm)
```

```
## Warning: package 'tm' was built under R version 3.4.2
```

```
## Loading required package: NLP
```

```
library(wordcloud)
```

```
## Loading required package: RColorBrewer
```

```
library(memoise)
```

Next, you will need to create some input text files to read and use for the word cloud. In this post we will be using books from Project Gutenberg, for example, Frankenstein at <http://www.gutenberg.org/cache/epub/84/pg84.txt>. Simply copy the text from the book and save it as a .txt file, removing any unnecessary text along the way. If you get a (incomplete final line found on './pride.txt') warning, make sure your .txt file ends with a blank line.

After creating the book .txt files, you will create a list of books available for the user to use in word clouds. In the list function, define each element as "Title" = "filename", for example since we saved Frankenstein as frankenstein.txt, we should have "Frankenstein" = "frankenstein". If prompted, save it as a UTF-8 encoding. Below is the list we will create for this post.

```
books <- list("Frankenstein" = "frankenstein",
              "Pride and Prejudice" = "pride",
              "The Adventures of Sherlock Holmes" = "sherlock",
              "Dracula" = "dracula")
```

Next, we will create a getWords function that will take in a book title and return a matrix of words and the counts of those words. All of the following code will be under the getWords function. First, we will need to read the lines in the text file and create a vector of the lines of text. To do this, we can use the functions "readLines()" and "sprintf()". Using sprintf() you can create the title of the text file, and then you can read it with the readLines function. Save this vector as "text".

You should get something like this

```
#ignore this line
book <- "frankenstein"
#stop ignore

#what you should get
text <- readLines(sprintf("./%s.txt", book),
                     encoding="UTF-8")
```

Next, we need to create a Corpus: a collection of texts, from the text vector. Using the tm function Corpus and VectorSource since text is a vector, we can create a collection of all the words in the book. Along the way, we can incorporate the functions removeNumbers, removePunctuation, removeWords, etc. to only select the words that we want. We can also use functions such as content_transformer to change the words to how we want them, in this case lowercase.

Example:

```
myCorpus <- Corpus(VectorSource(text))
#make lower case
myCorpus <- tm_map(myCorpus, content_transformer(tolower))
#remove punctuation
myCorpus <- tm_map(myCorpus, removePunctuation)
#remove numbers
myCorpus <- tm_map(myCorpus, removeNumbers)
#remove specific words
myCorpus <- tm_map(myCorpus, removeWords,
                  c(stopwords("SMART"), "thy", "thou", "thee",
                    "the", "and", "but"))
```

The next step is to use the corpus to create the matrix that the function getWords would return. To do this you will need to use the TermDocumentMatrix and as.matrix functions.

Example:

```
m <- as.matrix(TermDocumentMatrix(myCorpus,
                                   control = list(minWordLength = 1)))
```

Sort the matrix by the counts of the words in decreasing order to make it easier for the word cloud to get the top results, and finally place it in a function. The final getWords function can look something like this:

```
getWords <- function(book) {
  text <- readLines(sprintf("./%s.txt", book),
                     encoding="UTF-8")

  myCorpus <- Corpus(VectorSource(text))
  myCorpus <- tm_map(myCorpus, content_transformer(tolower))
  myCorpus <- tm_map(myCorpus, removePunctuation)
  myCorpus <- tm_map(myCorpus, removeNumbers)
  myCorpus <- tm_map(myCorpus, removeWords,
                    c(stopwords("SMART"), "thy", "thou", "thee",
                      "the", "and", "but"))

  m <- as.matrix(TermDocumentMatrix(myCorpus,
                                    control = list(minWordLength = 1)))

  sort(rowSums(m), decreasing = TRUE)
}
```

Finally, we can also use the memoise function. Memoise takes a function and creates a cached copy. Essentially what memoise does is it saves any new calls of the function and re-uses old calls in order to speed up any function calls. This is especially useful in cases such as this one where we are reading large .txt files.

Example of memoise function use:

```
getWords <- memoise(function(book) {
  text <- readLines(sprintf("./%s.txt", book),
                     encoding="UTF-8")

  myCorpus <- Corpus(VectorSource(text))
  myCorpus <- tm_map(myCorpus, content_transformer(tolower))
  myCorpus <- tm_map(myCorpus, removePunctuation)
  myCorpus <- tm_map(myCorpus, removeNumbers)
  myCorpus <- tm_map(myCorpus, removeWords,
                    c(stopwords("SMART"), "thy", "thou", "thee", "the", "and", "but"))

  m <- as.matrix(TermDocumentMatrix(myCorpus,
                                    control = list(minWordLength = 1)))

  sort(rowSums(m), decreasing = TRUE)
})
```

Word Cloud Maker

```
library(shiny)
```

UI

Choose a book:

Frankenstein

Change

Minimum Frequency:

1

15

50

1

6

11

16

21

26

31

36

41

46

50

Maximum Number of Words:

1

50

1

6

11

16

21

26

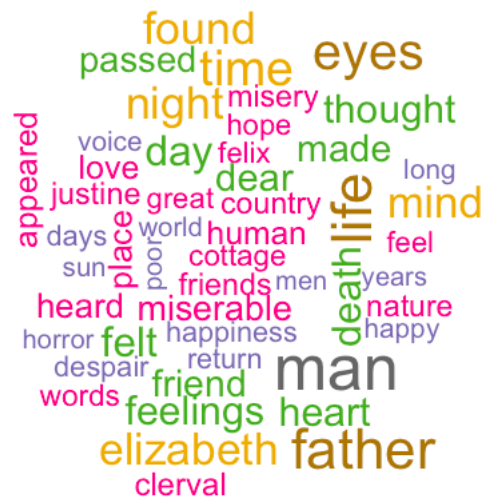
31

36

41

46

50



We want the final result to look something like this. Similarly to the homeworks and labs we have done in class, we will begin with defining a ui object. Everything should be similar to the histogram setup in Homework 4, with the exception of the `actionButton` function. The `actionButton` function will create a button for the user to press and will enact the purpose of the button. In this post we will use:

Change

<https://shiny.rstudio.com/reference/shiny/latest/builder.html>

```
ui <- fluidPage(
  titlePanel("Word Cloud"),

  sidebarLayout(
    sidebarPanel(
      selectInput("selection", "Choose a book:",
                  choices = books),
      actionButton("update", "Change"),
      hr(),
      sliderInput("freq",
                  "Minimum Frequency:",
                  min = 1, max = 50, value = 15),
      sliderInput("max",
                  "Maximum Number of Words:",
                  min = 1, max = 50, value = 100)
    ),

    mainPanel(
      plotOutput("plot")
    )
  )
)
```

The first thing we will do is create a reactive terms function that will take the selected book ("selection") and create the words matrix. Inside we will use the input\$update object and an isolate() function. The isolate function will make it so even after the selection is changed in the drop down list, the ui plot will not actually update until the update button is pressed.

Example:

```

terms <- reactive({
  input$update

  isolate({
    withProgress({
      setProgress(message = "Processing...")
      getWords(input$selection)
    })
  })
})

```

Next we will make a repeatable function `wordcloud_rep`, that takes in the `wordcloud` function and repeats it. This will then map every word in the cloud. Finally, the `output$plot` variable should be defined as a `wordcloud_rep` with all the appropriate parameters.

Example:

```

###ignore
plot <- c(1)
output <- data.frame(plot)
###stop ignore

wordcloud_rep <- repeatable(wordcloud)

output$plot <- renderPlot({
  v <- terms()
  wordcloud_rep(names(v), v, scale=c(4,0.5),
    min.freq = input$freq, max.words=input$max,
    colors=brewer.pal(8, "Dark2"))
})

```

Throw these above two examples into your server object to get something like this!

```

server <- function(input, output) {
  terms <- reactive({
    input$update

    isolate({
      withProgress({
        setProgress(message = "Processing...")
        getWords(input$selection)
      })
    })
  })

  wordcloud_rep <- repeatable(wordcloud)

  output$plot <- renderPlot({
    v <- terms()
    wordcloud_rep(names(v), v, scale=c(4,0.5),
      min.freq = input$freq, max.words=input$max,
      colors=brewer.pal(8, "Dark2"))
  })
}

```

Now you're all done! Run your shiny app by placing the line `shinyApp(ui = ui, server = server)` at the end of your `wordcloud.R` script and you're all done!

Conclusion

In class, we've seen a few applications of shiny app. However, just by browsing <https://www.rstudio.com/products/shiny/shiny-user-showcase/>, you can see that there are so many cool ways to use shiny. We've learned one more way to use it here, in a wordcloud, but there are plenty of more in-depth applications to create. In addition, we see that we can break down .txt files using text mining really simply. The text mining package (tm) has plenty more uses, and it shows that you can go beyond just reading and analyzing .csv or .xl files in R. To summarize, this post is meant to give a glimpse of what shiny can do and to encourage you to explore both text mining and shiny app further.

References

<https://shiny.rstudio.com/gallery/word-cloud.html>

<https://shiny.rstudio.com/articles/scoping.html>

<https://cran.r-project.org/web/packages/tm/vignettes/tm.pdf>

https://www.rdocumentation.org/packages/tm/versions/0.7-2/topics/tm_map

<https://cran.r-project.org/web/packages/wordcloud/wordcloud.pdf>

<https://cran.r-project.org/web/packages/memoise/index.html>

<https://www.rdocumentation.org/packages/memoise/versions/1.1.0/topics/memoise>

<http://www.gutenberg.org/files/1342/1342-0.txt>

<http://www.gutenberg.org/cache/epub/84/pg84.txt>

<http://www.gutenberg.org/cache/epub/345/pg345.txt>

