Introduction to Text Analytics

CSCI 3907/CSCI 6444

Introduction:

Text Analytics is what I consider one of the “Big Three” applications in Data Science. As we noted early in the course, perhaps 80-85% of human knowledge is still represented in text. Extracting that knowledge requires a disciplined approach that requires – in my view- a two part process: (1) analyzing the text and extracting words, phrases, sentences and concepts from it, (2) applying natural language processing (NLP) techniques to understand the syntax and semantics of sentences in order to produce knowledge for various applications.

This document is going to walk you through how to apply methods from some of the text analytics packages in R to extract data and information from text. You will apply these methods in Class Project 3.

Note that I use str(…) a lot in order to show you the structures that are generated by the different methods.

You should be using R 4.1.0 or later. Make sure you reload the packages mentioned below in order to get the latest versions.

I have selected a chapter from the volume *The Chemistry of Cookery* by W. Mattieu Williams, dated 1892, published by CHATTO & WINDUS, PICCADILLY, London. The title of this chapter is “The Boiling of Water” which I am sure you will find fascinating.

So, let’s begin with our analysis.

First, we need to set our directory. I have called it “Boiling” and the text document is “BoilingOfWater.txt” contained therein.

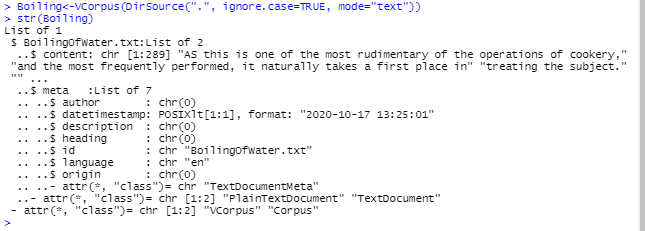


Install the package “tm” which contains some of the methods we need.



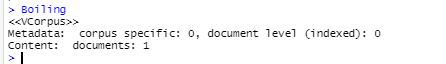
A VCorpus is a collection of documents. In our case, we will have just one document. A VCorpus is kept entirely in memory. VCorpus stands for Volatile Corpora.

The “.” means use the current directory, which we set by setwd(…). It will all .txt files into the VCorpus. We ignore case so that we don’t have to worry about and lower case. But, in your project you may have to consider case.

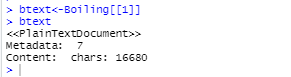


We use str(…) to examine the data structure produced. We see it has two components the text, consisting of 289 lines and the metadata – not much.

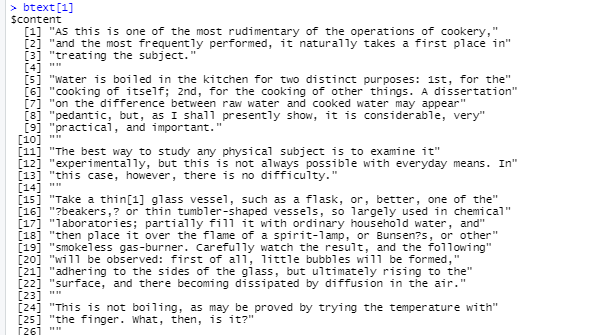
If we print Boiling, we get:



We extract the text from the corpus by:



The text contains 16680 characters. To see the content of the document:



The rest of the lines have been omitted.

…….

Hint:

**If** the document has identifiable chapters, you may use the which method to identify the indices for the chapters. For example:

indx\_ch1 <- which(book == “I.”, arr.ind=TRUE);

book\_chapter1 <- book[(indx\_ch1+1:indx\_ch2-1),];

to get the last chapter you would use:

book\_chapterN <- book[(indx\_chN+1:nrow(book),]

You can write the chapters of the book to separate files. For example,

dir.create(“chapters”);

write.table(book\_chapter1, file = “chapters/book\_chapter1.txt”, sep = “\t”, row.names=FALSE, col.names=FALSE,quote=FALSE);

……..

A *document term matrix* (DTM) is one of the most common formats for representing a text corpus (i.e. a collection of texts) in a bag-of-words format.

A DTM is a matrix in which rows are documents, columns are terms, and cells indicate how often each term occurred in each document.

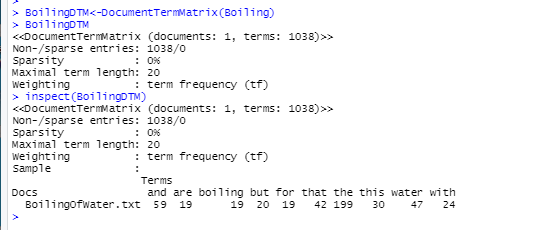
The advantage of this representation is that it allows the data to be analyzed with vector and matrix algebra, effectively moving from text to numbers.

A *term document matrix* (TDM) is a matrix in which rows are words and columns are documents.

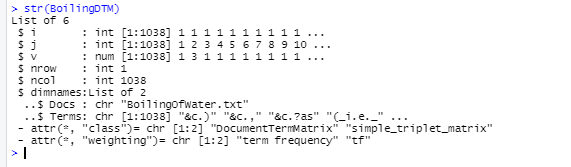
Two ways of showing the same information

But, sometimes, you want to process by words, other times by documents.

Compute the DocumentTermMatrix:

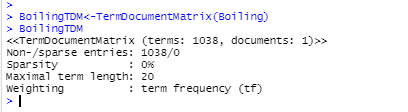


Looking at the structure, we see some fields:

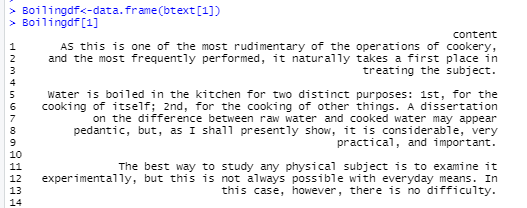


In this case, because we have a single document the number of rows is one. The number of columns represents the number of unique terms in the document.

And, we compute the TermDocumentMatrix:



So, lets make it a data frame:



Compare this to btext[1] above.

**Corpus Cleansing – Data Wrangling**

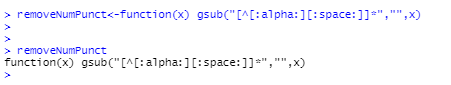
We already have the document in lower case.

There are 1038 terms. And we see some of the terms have punctuation and other special characters.

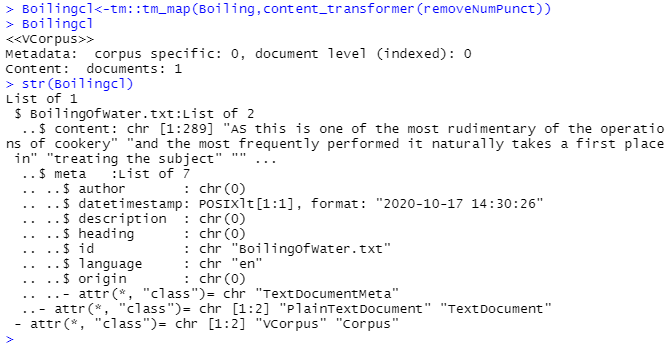
But, this is misleading because we left in punctuation, numbers, web addresses, etc. from the text. I some cases, you will need to retain numbers and punctuation (certainly for some NLP processing)

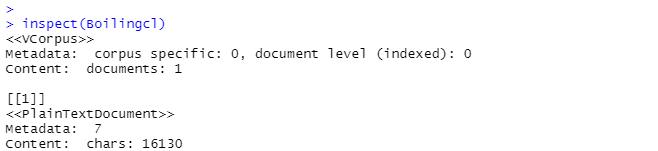
If the quote character (‘) appears in the text, you can use the **noquote**() function to remove them.

Define a function to remove numbers and punctuation:



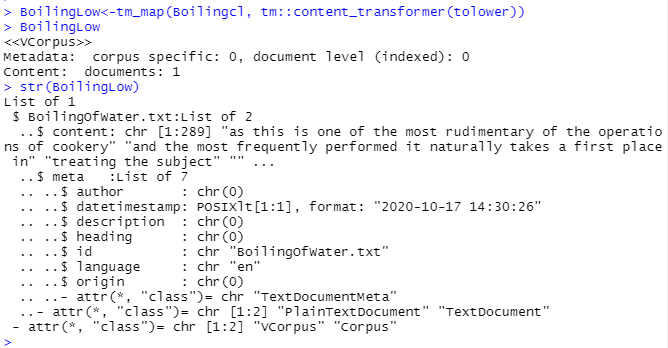
To remove numbers and punctuation, we create a clean corpus:

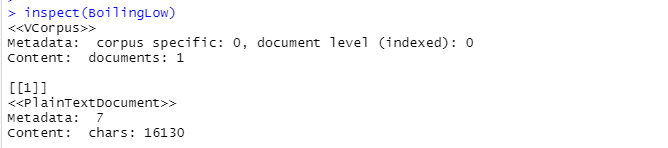




Note that we have removed some 550 characters from the corpus.

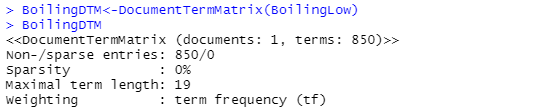
Also, let’s make sure everything is in lower case.

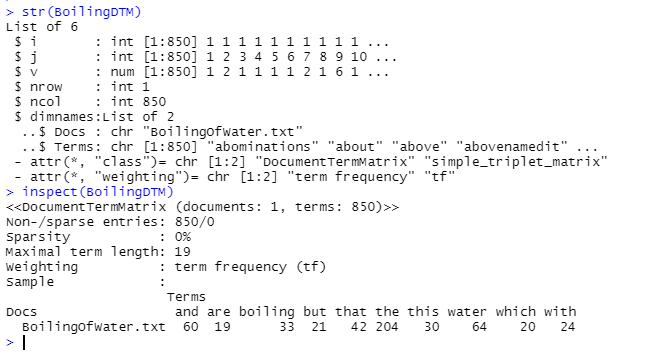




OK. Same number of characters now all in lower case.

**Compute the Document Term Matric (DTM)**





Why do I have sparsity as 0%?

A [document term matrix](https://www.rdocumentation.org/packages/tm/versions/0.7-7/topics/TermDocumentMatrix) is a matrix that has as rows the documents, as columns the terms, and 0 or 1 if the term is in the document in the row (1) or not (0).

Sparsity is an indicator that points out the "quantity of 0s" in document term matrix.

To make this clearer, here is an example from StackOverflow:

To understand those gists, let's have a look to a reproducible example that creates a situation similar to our problem:

library(tm)

text <- c("here some text")

corpus <- VCorpus(VectorSource(text))

DTM <- DocumentTermMatrix(corpus)

DTM

<<DocumentTermMatrix (documents: 1, terms: 3)>>

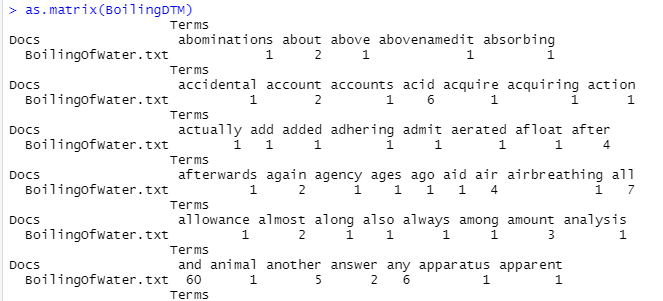
Non-/sparse entries: 3/0

Sparsity : 0%

Maximal term length: 4

Weighting : term frequency (tf)

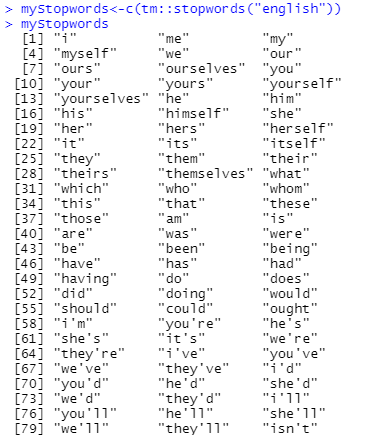
Looking at the output, we can see you have one document (so a DTM with that corpus is made of one line).



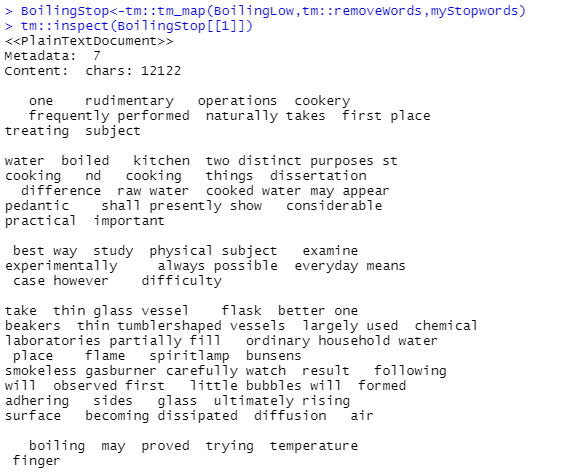
Rest omitted.

So, there are no sparse terms, e.g., “0’s” in the DTM.

So your sparsity is == 0%, because you cannot have some 0s in one document corpus; every term belongs to the unique document, so you'll have all ones or more if a term occurs more than once.:



…. Plus a lot more



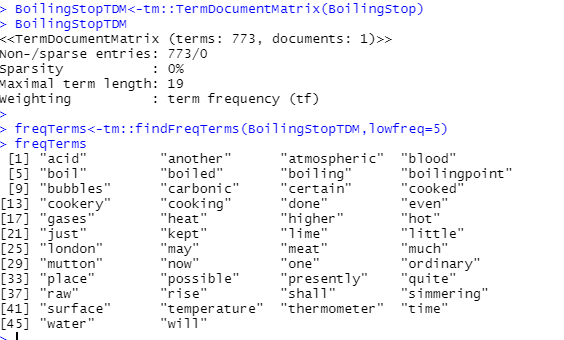
….. and more

So, we seem some words here we might like to remove: “may”, “one”, “nd”, etc…

This is left as a reader’s exercise.

We need to create TDM again now that we have removed the stop words.

Then, we can apply findFreqTerms with a lowFreq of 5, but you should experiment with different finstall.packages(worequencies.



To find the length of a string in freqTerms, we can do:

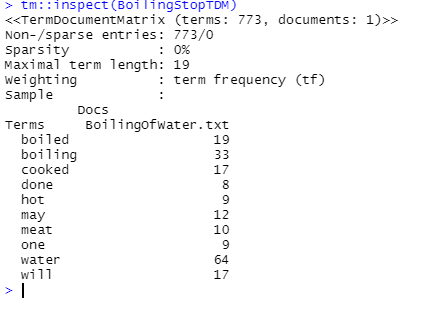


To find the lengths of all strings, use BoilingStopTDM, and write a loop that accesses each word, computers its length via nchar(…), then puts it in a vector.

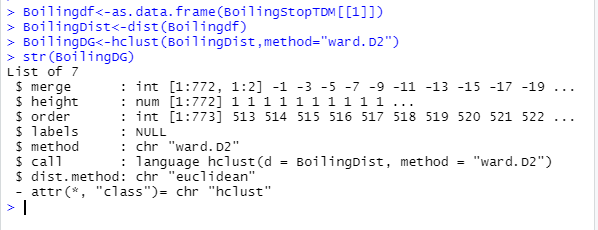
We can look at the term frequency using termFreq:



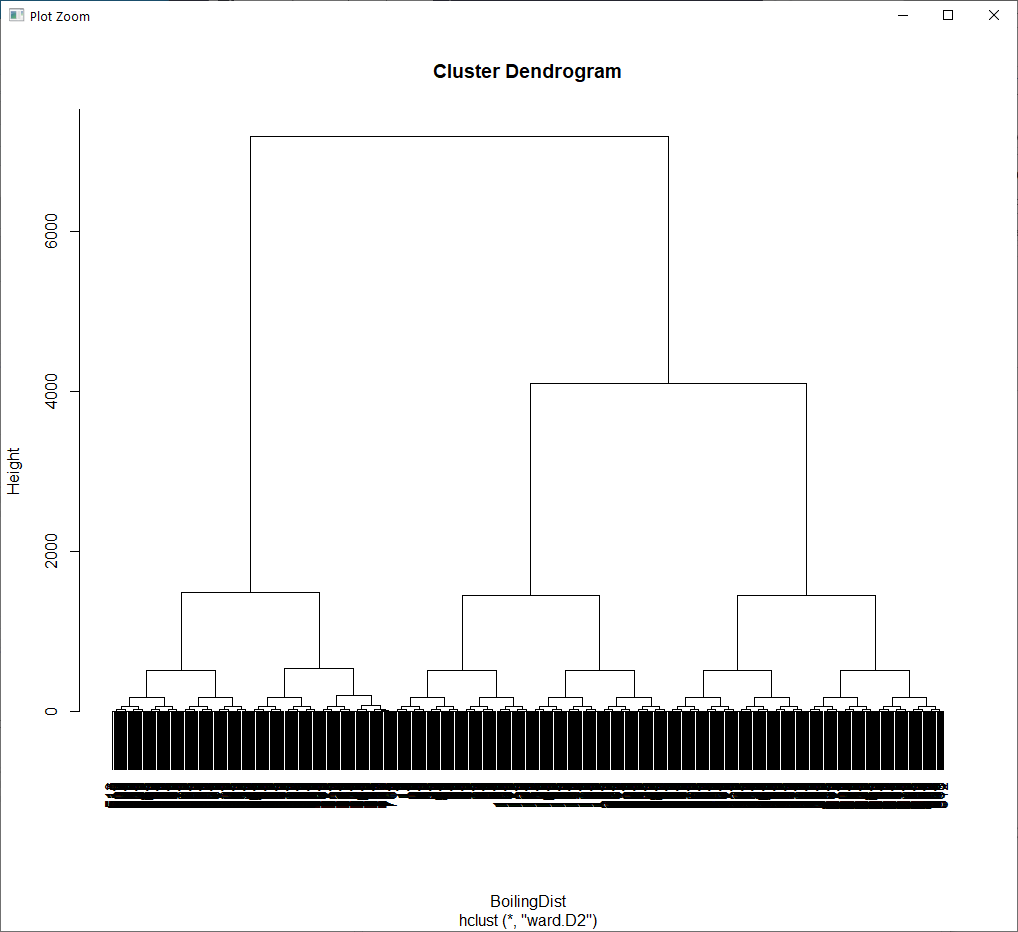
Let’s inspect BoilingStopTDM again:



Now, let’s draw a dendrogram:

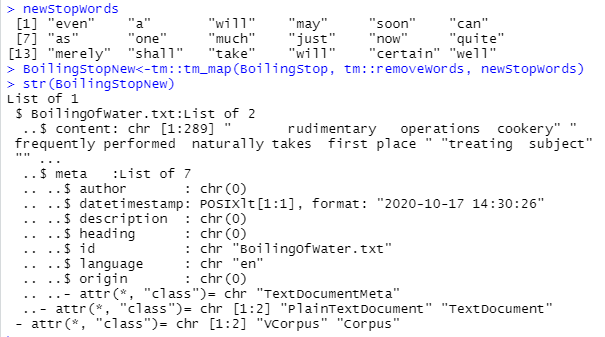
BoilingNewDF



B

Not really helpful.

Let’s try to eliminate some more sparse words:



So, create a new TDM, data frame, and distance matrix.



Produces much of the same thing.

So, you will probably have to eliminate more words to get a good dendrogram.

See what you get with the test document, then pick high sparsity to remove number of words.

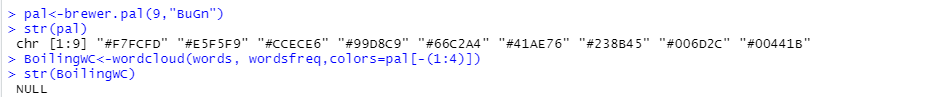
**Word Cloud:**

Install package wordcloud, make it a library:

Let’s get the set of words from Boiling:



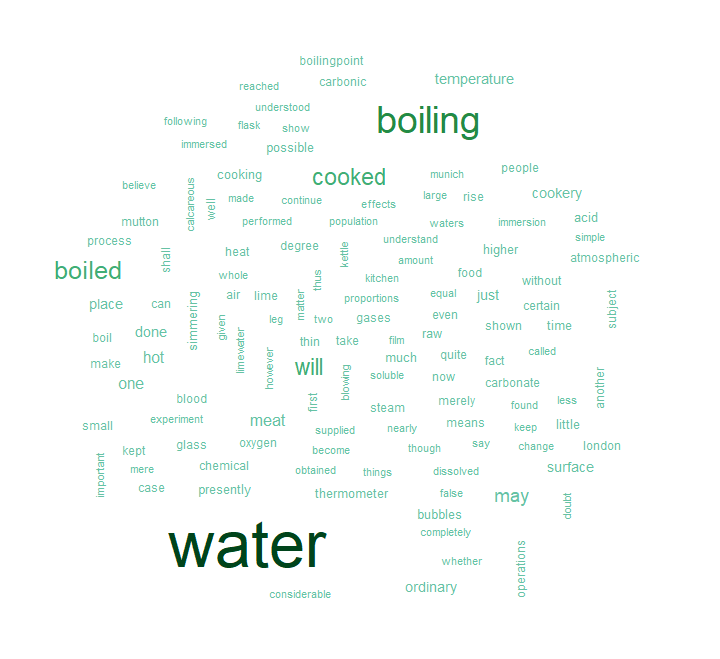
And, the frequencies are in Boilingtf. Set wordfreqs = Boilingtf.



**Note: str(BoilingWC) returns NULL because this method only displays to the screen. This is how the author wrote the code.**

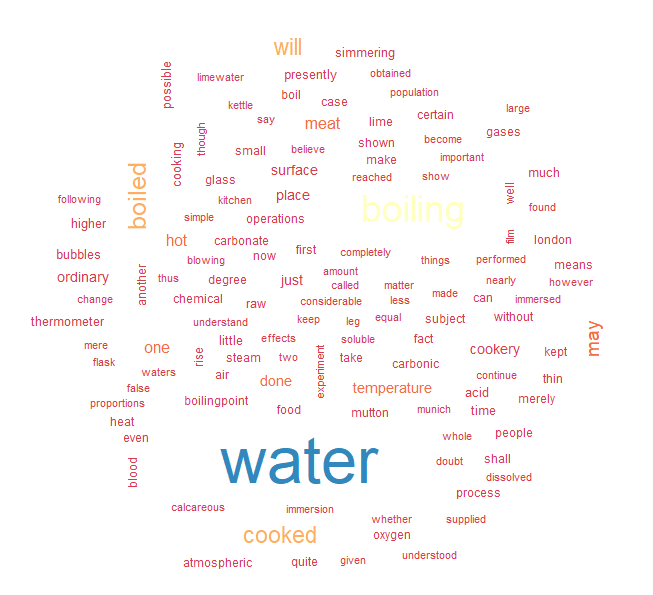
Pal is the palette of colors. See the ‘brewer’ documention via RStudio Help.

The word cloud is:



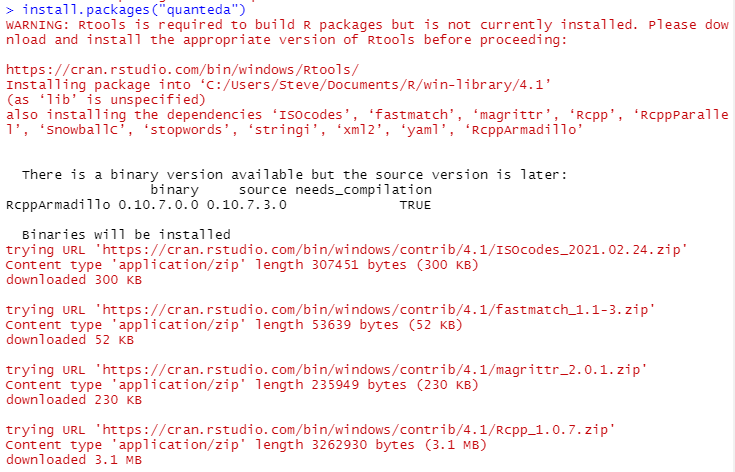
And, with a different pallet:





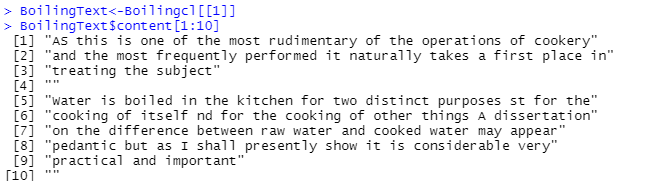
Note: wordcloud displays the result, but does not return a structure, hence the NULL above.

Install “quanteda”, and make it a library:



Plus more lines omitted.

Let’s look at first few lines of Boilingcl – using the clean copy



Let’s apply some tokenization:



I only did 10 lines so I could show you the structure. There are many more lines you will see in the structure when you do the whole document using BoilingText$content.

Note that chr(0) indicates a blank line above.

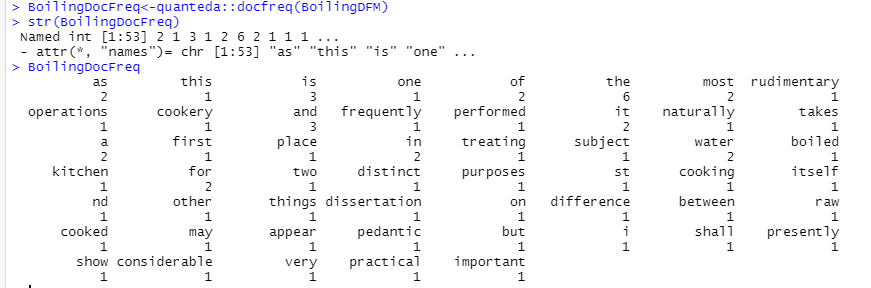
You may want to write an R function to delete blank lines.

Now, dfm(…) constructs a sparse document-feature matrix, so let’s do it:

Note that I am using the 10-line BoilingTokens so it fits on a page.



Now, let’s get the frequency of terms in the dfm:



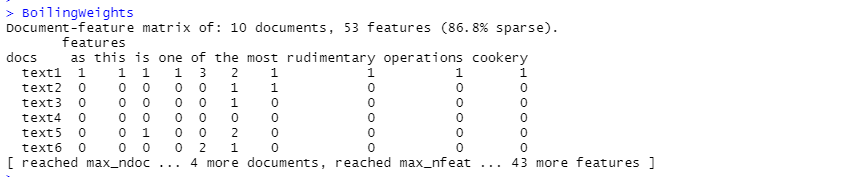
Note: we should have removed the stop words, but did not.

You can do this when you create the DFM by asserting the remove keyword as shown in the quanteda documentation.

Now, let’s assign weights to these words:



See the documentation for dfm\_weight for options.



The ten documents are the ten lines we used with the default labeling of text1

Obviously when you do the entire document that will be your test case, you will see more definitive data.

Now, let’s compute the tf-idf score.

dfm\_tfidf weights a dfm by term frequency-inverse document frequency (tf-idf), with full control over options. Uses fully sparse methods for efficiency.

See the quanteda documentation for options.



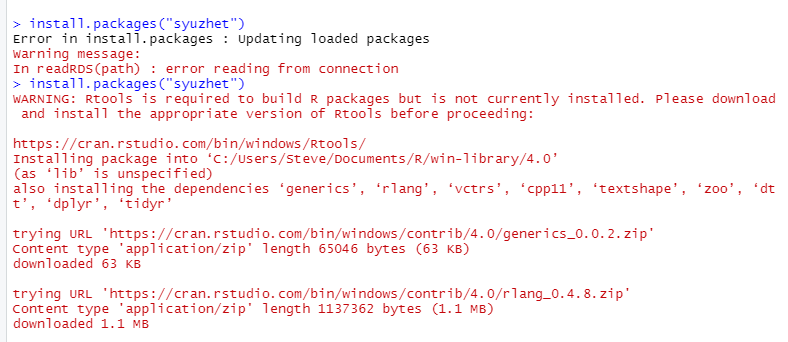
Note: dfm\_tfidf calls both dfm\_weight and docfreq internally, but, I wanted to show you what they compute.

Note about Packages: Many packages try to be self-contained. This means that you will see different versions of the same functions repeated in different packages. For example, quanteda has both a “tolower” and a “toupper” function defined.

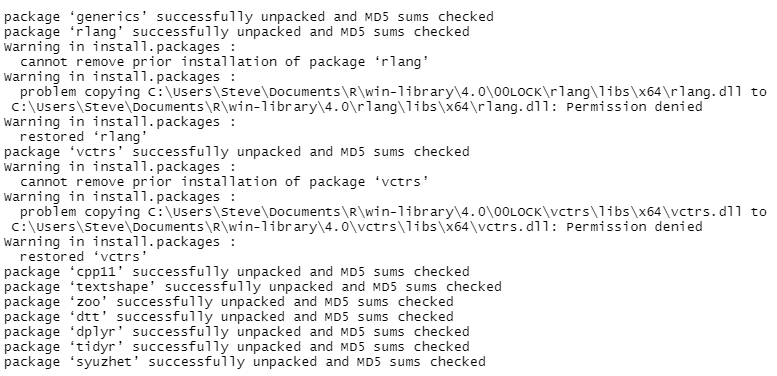
This is probably a good idea when you are writing code where you don’t have access to source code of other packages that you will use. The documentation may be very good, partially complete, or almost totally missing. Rarely do you see documentation that is very good – even R packages have some lapses.

**AXIOM: The ground truth is not the documentation of a package, but the source code of a package.**

Install package “syuzhet”



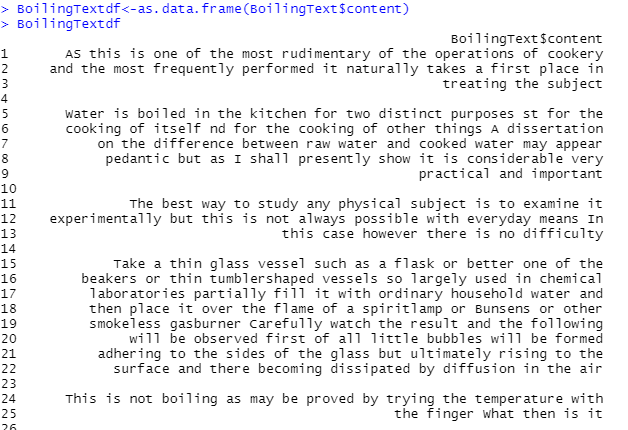
<<Skip a whole bunch of messages>>

Jtext=jtext[1]

mode=mode=”jtext=jtext=jekyllWe see “syuzhet” depends on a number of other packages.

Isn’t it nice that R automatically loads dependent packages for you?

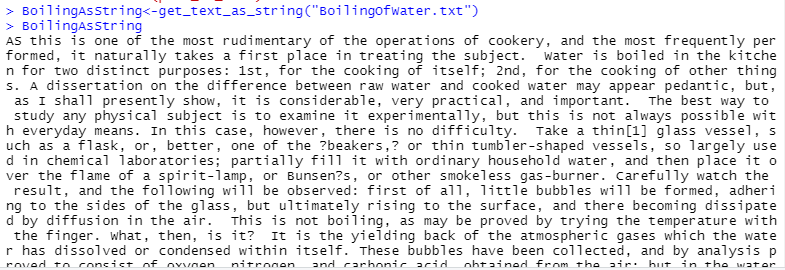
Extract the text as a data frame:



<<plus 260+ more lines>>

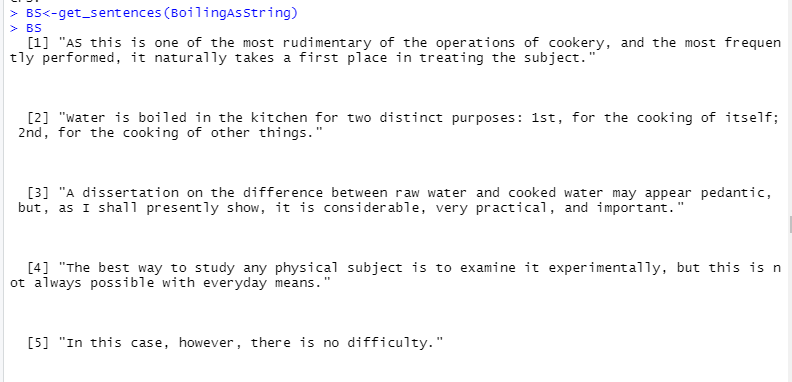
Let’s get the sentences in BoilingOfWater.txt

First, read the file as one large string.



Many lines omitted.

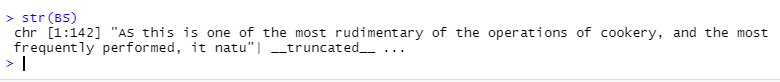
Now, get the sentences:



So, get\_sentences parses the text string into individual sentences.

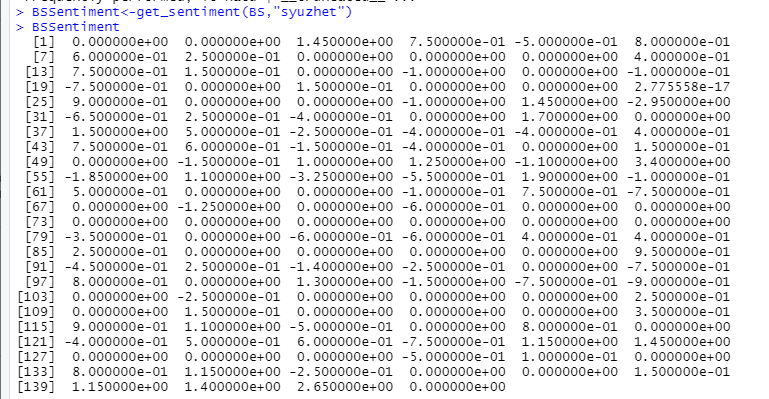
After you have collected the sentences or word tokens from a text into a vector, you will send them to the get\_sentiment function which will assess the sentiment of each word or sentence. This function takes two arguments: a character vector (of sentences or words) and a “method.” The method you select determines which of the four available sentiment extraction methods to employ.

In RStudio in the lower right pane, you can select “Help”, then type get\_sentiment in the search box to see the documentation.

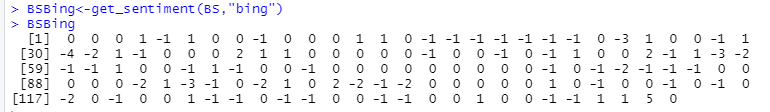


Note that BS is a character vector.

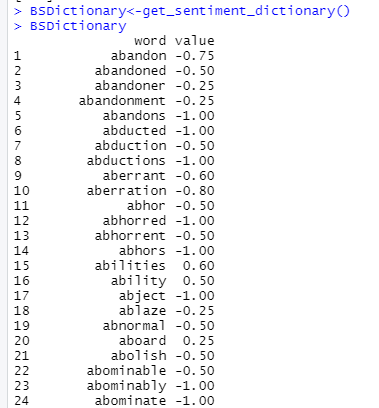
Let’s call get\_sentiment with the default value of “syuzhet”:



So, it returns the sentiment for each sentence in BS. The different methods will return different results.

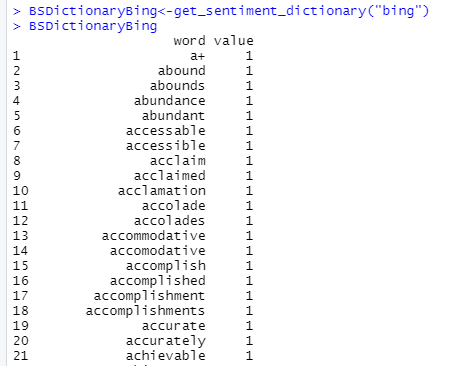


Let’s look at the sentiment dictionary for “syuzhet”:



And, there are about 11,000 more rows.

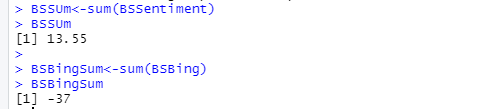
Here is the first part of the Bing Dictionary:



And about 6500 more rows.

Notice that the two dictionaries have different lengths because they are based on different methods and different samples of extracts.

Sum the values of the sentiment vector in order to get a measure of the overall emotional valence in the text:

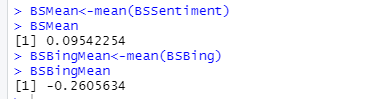


Hmmm! Different values of emotional content.

So, what you need to do when doing sentiment analysis is make sure you understand the basis for the sentiment dictionary that you are going to use. Otherwise, you may get misleading results.

So, syuzhet dictionary says BoilingOfWater had a positive content, where as BSBing says it has a negative emotional content.

Let’s look at the means:



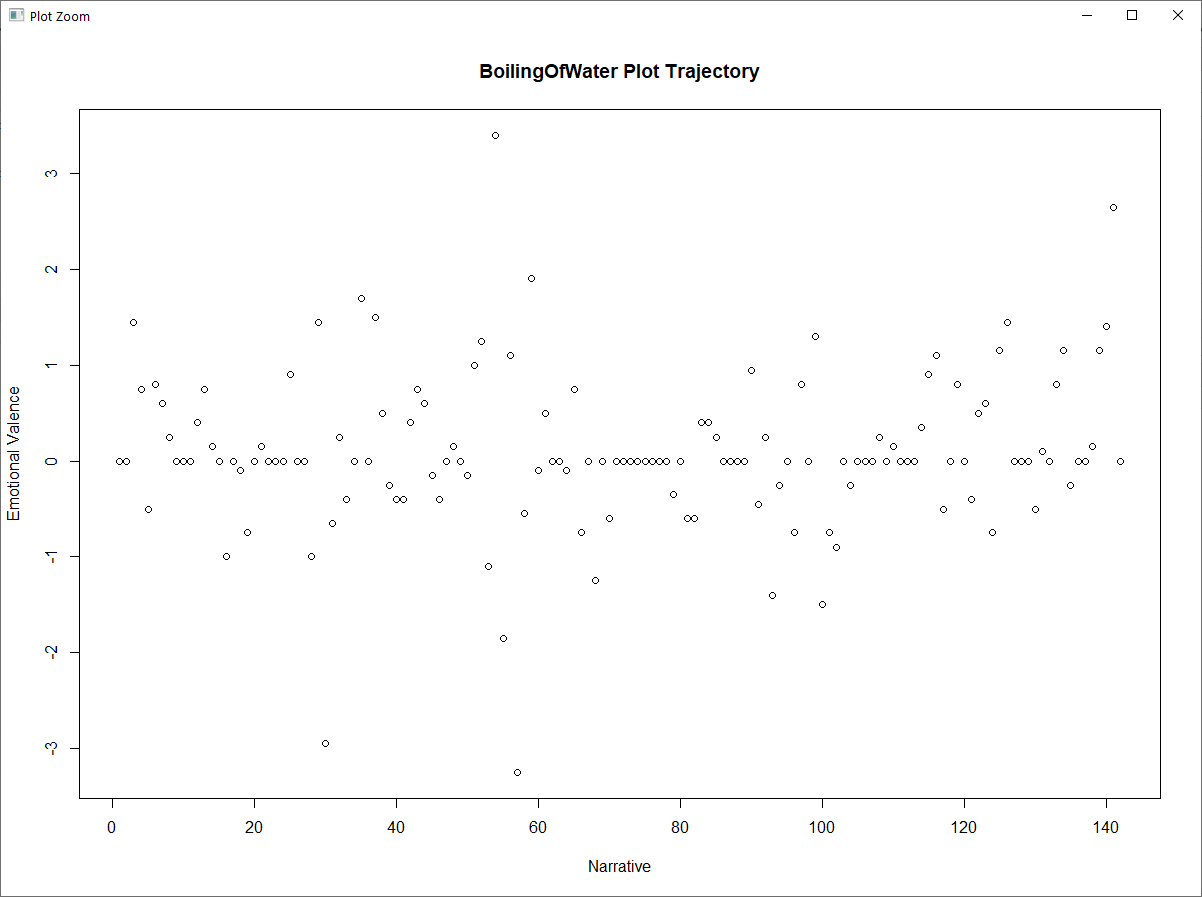
You can use the summary(…) function to see how the sentiments are distributed for each case.

You should also do this using the method get\_nrc\_sentiment(…)

While these global measures of sentiment can be informative, they tell us very little in terms of how the narrative is structured and how these positive and negative sentiments are activated across the text. You may, therefore, find it useful to plot the values in a graph where the x-axis represents the passage of time from the beginning to the end of the text, and the y-axis measures the degrees of positive and negative sentiment.

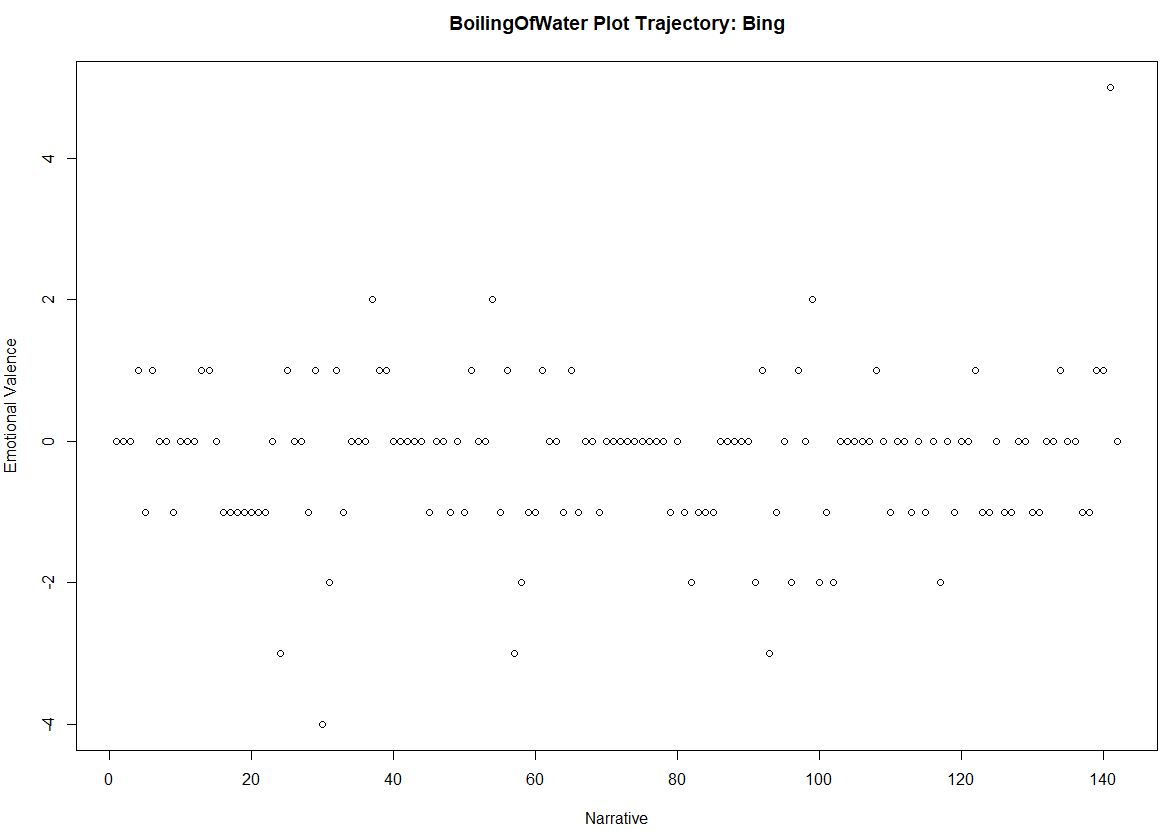
Now, for BoilingOfWater, this won’t make much sense, unless you can get very enthused about boiling of water (some people do!!):



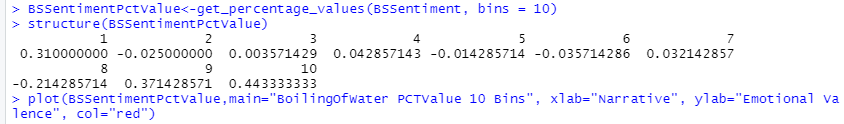


And, for BSBing:

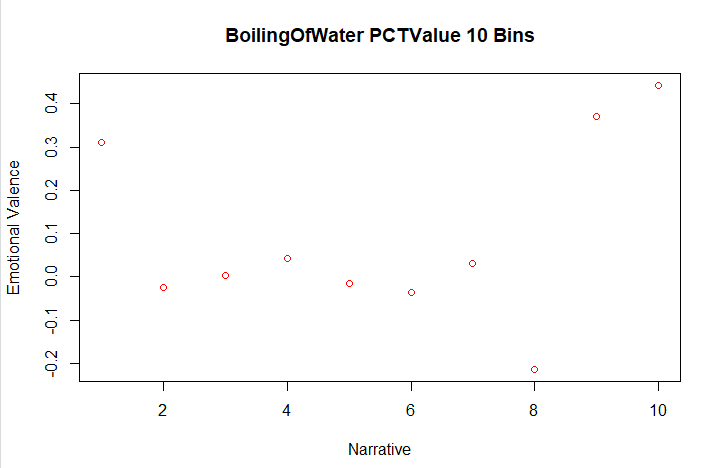




When it comes to comparing the shape of one trajectory to another, the get\_percentage\_values function can be useful. The get\_percentage\_values function divides a text into an equal number of “chunks” and then calculates the mean sentiment valence for each.

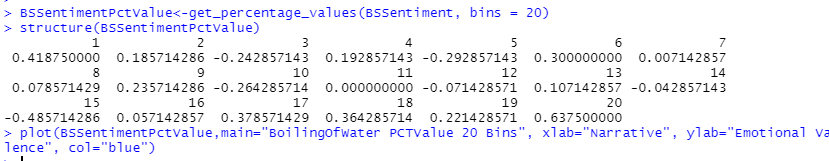


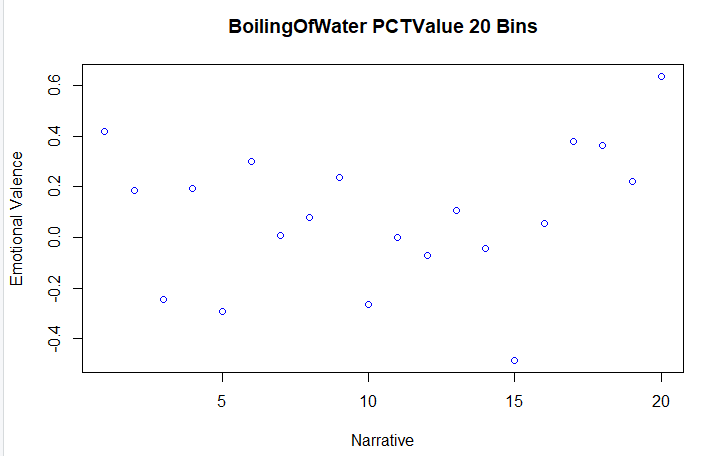
And, the plot looks like:



Wow!! Who would have expected such emotional intensity about boiling water at the end!

Now, with 20 bins!





More granularity gives a better view of emotional jags about boiling water.

When a series of sentence values are combined into a larger chunk using a percentage based measure, extremes of emotional valence tend to get watered down. This is especially true when the segments of text that percentage based chunking returns are especially large. When averaged, a passage of 1000 sentences is far more likely to contain a wide range of values than a 100 sentence passage. Indeed, the means of longer passages tend to converge toward 0.