Intro to Quantitative Textual Analysis

Alex Leslie

4/4/2019

Contents

Welcome to this beginner workshop on quantitative textual analysis! This workshop will cover some of the basic forms of analysis we can carry out on a single document (in this case, a book), but it will also serve as a basic introduction to R for those totally unfamiliar.[1]

First of all, run the section of setup code above by clicking the green arrow in the upper right of the grey box. The code for this workshop relies on a number of different packages in order to help demonstrate a variety of different approaches to textual analysis in R. Packages are a common aspect of coding in R: they contain user-written functions designed to make analysis faster and easier.

If you'd like, go to the Gutenberg Project website and find a text to work with - any text, literary or nonliterary, but it should be a text you're already familiar with. If the choice is too much pressure, feel free to use one of my samples. The default for this workshop is George Washington Cable's important 1879 American novel *The Grandissimes*, but I've included a number of other texts to choose from.

Getting Started in RStudio

Don't panic: you're in RStudio Cloud, an environment that makes coding in R more transparent. Let's take a brief tour. Your window is divided into four segments. In the upper right, you'll see the environment: this displays all the objects you currently have loaded into memory. In the upper left, you'll see the script editor: this is the place to work on code that you're currently writing (or borrowing from elsewhere!). To run code in code chunks (the grey areas that start with "“‘{r}”), you can either press Ctrl+Enter to run single lines or click the green arrow to run the entire chunk (Ctrl+Shift+Enter will do the same). In the lower left, you'll see the console: this is where code actually executes and where the output prints. In the lower right, you'll see a few different things. The "Files" tab shows whatever files you've uploaded to the RStudio Cloud; if you run any plots, they'll show up in the "Plots" tab; you can also get help in the "Help" tab.

Programming languages like R distinguish between several data types; these include numeric (e.g. 1, 2, 3), boolean (e.g. TRUE, FALSE), and character (e.g. "n", "s", "v"). Quantitative textual analysis is primarily concerned with characters, but we'll sometimes rely on the others in the process of working with characters. A string of characters (e.g. "string", "characters") are called a character string.

First, we need to read in our data into our environment. We'll do that by using a function, `readLines`, to read in a .txt file and assign that data (using `<-`) to an object with whatever name we'd like ("gutenberg_file").

[1] This workshop is of course influenced by Matthew Jockers' classic *Moby-Dick* workshop, one form of which can be found on his website. It is also influenced by Julia Silge and David Robinson's *Text Mining with R*, which I recommend in particular for anyone looking for a more sustained introduction to quantitative textual analysis.

Since we're passing two parameters to `readLines` - the file path to the .txt file and the desired character encoding - we need to separate them with a comma (`,`). These notations constitute the most basic grammar of R.

```
gutenberg_file <- readLines("sample_texts/grandissimes.txt", encoding = "UTF-8")
```

Type "gutenberg_file" into the console and click enter in order to see what this object actually looks like. You'll note we have a series of lines, each enclosed in quotation marks and each with a number on the left. Each of these lines is called a *character string*; together, they form what we call a *vector*.

If you scroll up in the console, you'll see that the first twenty or so lines are all Gutenberg Project data and the next chunk of lines are metadata from the book itself (copyright, title page, table of contents, etc). Let's get rid of this extra stuff so that we're working just with the body of the text.

Preparing the Text

Find what you want to be the first line included in the text, and enter that where I've written "CHAPTER I". Then edit the next line of code so that it reflects the text you're working with. All Project Gutenberg texts end with a formula line that reads "End of Project Gutenberg's [title], by [author]", so you'll just want to double check to make sure you enter the title and author as they appear in the file.

The `which` function will allow us to identify which elements of a vector exactly match whichever character string we're looking for.

```
which(gutenberg_file=="CHAPTER I")
```

```
## [1] 164
```

What is this number? It's the position in the gutenberg_file vector of the elements (in this case just one) containing the character string "CHAPTER I".

We can test this by *indexing* the gutenberg_file vector using brackets, `[]`, to select only the element(s) in the desired position(s). Indexing is one of the most useful and versitile tools in R, so you'll encounter it frequently today: it simply designates a subset of whichever object you're working with. For example:

```
gutenberg_file[164]
```

```
## [1] "CHAPTER I"
```

The 164th element of gutenberg_file is indeed "CHAPTER I".

We'll save this and the posiion of the desired end line as objects into memory.

```
start <- which(gutenberg_file=="CHAPTER I")
end <- which(gutenberg_file==
              "End of Project Gutenberg's The Grandissimes, by George Washington Cable")
```

You can index as many positions as you'd like. Next we'll index gutenberg_file so that we're only selecting the elements from the start position through (`:`) the end position minus one (since we don't want the "End of" line itself).

```
text <- gutenberg_file[start:(end-1)]
```

Give indexing a try on your own: in the console, write the short line of code to return the first ten elements of our new text vector.

As you can see, there's a bit more tidying to do. The lines format was useful for making those initial trims, but it's not really a subdivision that we're interested in (unlike, say, individual words or chapters). So let's paste all of those lines of characters together to make one big character string, separating each with a blank. This will effectively remove the blank line null characters `""`.

```
text <- paste(text, collapse=" ")
```

Given that many searches are case-sensitive by default, it's easier to work with text when it's all in lower case.

```
text <- tolower(text)
```

Next we'll split that one big character string into a big vector in which each word is its own separate element. By splitting our string (with `strsplit`) into its smallest units we'll be able to maximize our options for analysis. The second parameter in `strsplit` is what's called a regular expression: a metacharacter that, in this case, refers to any non-word characters.

```
word_vector <- strsplit(text, "\\W+")
word_vector <- unlist(word_vector)
```

The result should be a vector of character strings, one for each word in our text. With this we can begin some analysis.

Text Composition and Word Usage

How many unique words are there in *The Grandissimes*? In other words, what is the `length` of the vector of `unique` elements in word_vector? We'll wrap the first function we want R to carry out, `unique`, in the second, `length`. This is called nesting functions.

```
length(unique(word_vector))
```

```
## [1] 11606
```

How many times is each word in the text used? The handy `table` function will automatically tally this up. Since there are a lot of words, we'll index only the first twenty elements of the resulting table.[2]

```
table(word_vector)[1:20]
```

```
## word_vector
##            _           _a           _à    _abandon_     _absolvo
##           71            2            1            1            1
##          _ad       _again          _ah         _aie       _allez
##            1            1            9            2            2
##     _always_         _ama  _américain_ _américains_          _an
##            2            1            2            1            1
##        _and_      _animal    _appelez_    _apportez       _aqua_
##            1            1            1            1            1
```

This will be easier to read if we `sort` the results of the `table` function according to total use rather than the default alphabetical order. Sort takes a second variable, `decreasing=` TRUE or FALSE, so it needs a comma separating this information from the object actually being sorted. We can again index only the top 20 elements, to keep things simple.

[2] What are all those underscores doing? This is the way that this particular transcription of *The Grandissimes* marks dialect speech or phrases. For this reason we may wish to keep them - if we were looking at dialect in the novel, say. But we might prefer them removed. The `gsub` function will substitute whatever character pattern we want to replace with whatever character pattern we want to replace it with:

```
word_vector <- gsub("_", "", word_vector)
```

This solves the issue for most words, but all elements that were nothing but an underscore will now be a null character, `""`. So we'll also need to index all elements of word_vector `which` are not (`!=`) null characters:

```
word_vector <- word_vector[which(word_vector!="")]
```

```r
sorted_words <- sort(table(word_vector), decreasing=TRUE)

sorted_words[1:20]
```

```
## word_vector
##  the  and   of    a   to   in   he  his  was that   it  you    i with  her
## 6789 3385 3378 2813 2796 1770 1508 1387 1348 1229 1219 1212 1074 1068 1025
##   as  not  had   is  but
##  852  826  808  808  789
```

Unsurprisingly, the most commonly used words in most texts are, well, a bit boring. Let's try indexing only the strings in word_vector `which` contain a number of characters (`nchar`) greater than three.

```r
top_words <- word_vector[which(nchar(word_vector) > 3)]
sort(table(top_words), decreasing=TRUE)[1:20]
```

```
## top_words
##        that        with        said  frowenfeld        have        this
##        1229        1068         525         479         456         426
##        they        from       which       there grandissime       would
##         419         409         391         341         340         318
##      honoré        were       their        into        upon        what
##         314         309         297         272         272         259
##        them        will
##         257         257
```

That helped, but there's still a fair bit of static here. This time, we'll use a stop list - a list of common words that we don't want gumming up the works - to cull a bit more (there are plenty of stoplists lying about online; I've pulled mine from the Princeton CS website). Then we'll index top_words to exclude (with the negative operator `!`) all of the elements of top_words that are also elements `%in%` the stoplist.

```r
stoplist <- readLines("https://algs4.cs.princeton.edu/35applications/stopwords.txt")
top_words <- top_words[! top_words %in% stoplist]

sort(table(top_words), decreasing=TRUE)[1:30]
```

```
## top_words
##  frowenfeld grandissime      honoré      aurora    clotilde    agricola
##         479         340         314         216         207         189
##        eyes      creole        back      doctor        made  apothecary
##         186         170         169         166         164         161
##        face        hand      turned        time     palmyre      joseph
##         160         156         150         149         142         139
##        head       raoul        good      moment        make        door
##         132         129         126         122         107         106
##       coupé        room        bras       great       keene       hands
##         101         100          97          97          97          96
```

Much nicer: this has eliminated some real words, yes, but it has also made the results more semantically meaningful. In the results for *The Grandissimes* we now have a lot of character names (the first six, in fact), but we can also see that the text is invested in temporality and fleetingness (time, moment, turned), physical features (eyes, hand, face), and magnitude (good, great). Any words that stand out here could make good candidates for future analysis.

This information can be visualized as a word cloud as well.

To do so, we'll need to save top_words as a `data.frame` instead. A *data frame* is another data type in R: it is basically the equivalent of a spreadsheet in that it contains observations (or rows) and variables (or columns). Our data frame, word_df, simply consists of a single variable or column made out of top_words, which I've called word.

```
word_df <- data.frame(word=top_words)
```

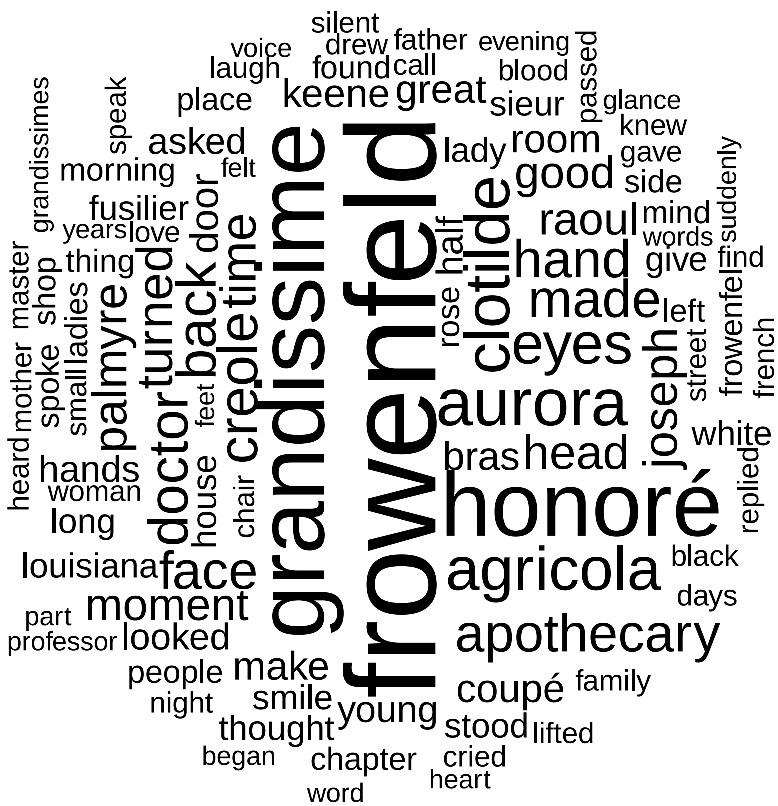
Next, we'll use the piping operator `%>%`, which simply funnels the results of one function directly into the next, making our code clearer and more concise. This allows us to take our new word_df data frame, `group_by` word, and `summarize` the number of observations in each group (i.e., the number of occurences for each word) as a new variable called total.

```
word_df <- word_df %>%
  group_by(word) %>%
  summarize(total=n())
```

To see what this actually looks like, click on word_df in the environment; this will launch a data viewer window in the script editor. (When you're done looking you can either close this window or click back to the workshop tab.)

Finally, we'll generate the wordcloud itself. The second line of parameters is all unique to the `wordcloud` function. If you look at the first line of parameters, though, you'll see that the `$` operator allows us to refer to just one variable or column of a data frame.

```
wordcloud(word_df$word, word_df$total,
          min.freq=5, max.words = 100, random.order=FALSE, rot.per=.35)
```



Word clouds might not be the most informational, but they sure do look fun.

There are a number of other dimensions of a text that can be quantified based on broad patterns of word use. We might begin to consider how the text represents gender, for example, by looking at its use of pronouns. Since sorted_words is in table format, this can be done simply by indexing the character string in question.

```r
sorted_words["he"]
```

```
##   he
## 1508
```

```r
sorted_words["she"]
```

```
## she
## 786
```

```r
sorted_words["his"]
```

```
##  his
## 1387
```

```r
sorted_words["her"]
```

```
##  her
## 1025
```

In the case of *The Grandissimes*, the masculine first person pronoun is clearly used more than the feminine first person pronoun, but the gap is closed when it comes to possessive pronouns. This is often the case in a novel with romance plots whose central protagonists are predominantly men. We might explore this further by finding all the words that come after "her," for example, to find out what this possessive pronoun most often possesses (ex., is the novel more interested in "her eyes" or "her resolve"?).

What about tense? English verbs aren't differentiated quite as reliably by suffix as verbs in other languages, but we can at least get a sense based on usage of basic "to be" conjugations.

```r
sorted_words["was"]
```

```
##  was
## 1348
```

```r
sorted_words["were"]
```

```
## were
##  309
```

```r
sorted_words["is"]
```

```
##  is
## 808
```

```r
sorted_words["are"]
```

```
## are
## 235
```

```r
sorted_words["am"]
```

```
## am
## 90
```

```r
sorted_words["will"]
```

```
## will
##  257
```

The *The Grandissimes* clearly spends more time in the past tense, which makes sense for a novel with such a deep investment in history. Texts can be skewed more towards one tense than another based on several factors like genre, subject, or time of composition (several twentieth-century fiction styles use present tense more often than their nineteenth-century precursors).

Depending on the focus of analysis, one may wish to compare some of these attributes across a corpus of texts, whether to identify unknown texts with particular features, to classify groups of texts, or to identify some kind of pattern.

Word Positions and Contexts

Let's focus on a single important word. You may choose an important character, concept, descriptor, one of the most frequently used words that stood out, anything. To demonstrate, I'll use the second half of the name Bras Coupé, an enslaved person in *The Grandissimes* whose curse on Louisiana's indigo plantations disrupts the novel. Finding the instances of a desired word is easy enough: just a simple `which` function.[3]

```
keyword <- which(word_vector=="coupé")
```

How about a basic calculation: what percentage of the total number of words (i.e., the `length` of the word_vector) are the keyword just searched for? To do this we'll nest a few functions, so keep track of the parentheses. First we'll divide to come up with the proportion, then multiply by 100 to get a percentage. To make things look nice, we'll `round` that number to 4 digits and `paste` a percentage sign after, clarifying that we don't want anything (`""`) `sep`arating what's being pasted together.

```
paste(round(length(keyword) / length(word_vector) * 100, 4), "%", sep="")
```

```
## [1] "0.0875%"
```

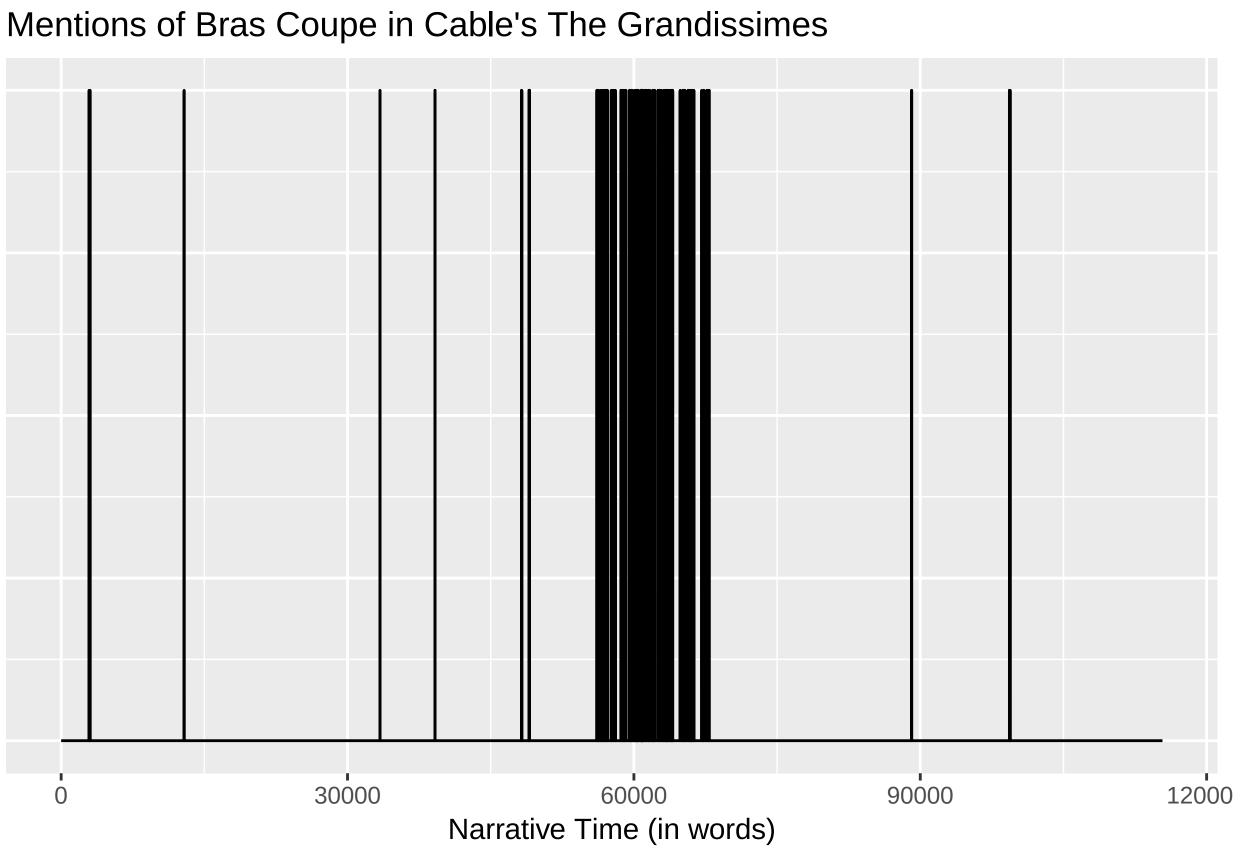
Of course, words aren't used evenly throughout a text: they often follow patterns of use, and these patterns might be of importance. In a fictional text, we might call this narrative time, that is, the time it takes the narrative to unfold, as opposed to the time in which the plot unfolds. There are several potential ways we might measure narrative time - for example, our unit of measurement might be chapters. Given that we already know the total number of words and the position of any given word within that total, we'll use words as our unit of measurement instead.

First we generate an x axis that is as long as there are words in our text. Then we make a y axis of the same length, where each value is 0 – except for the positions of our keyword, which we'll assign a value of 1. Finally, we save this as a data frame so that the two vectors match up with each other as two columns. That's all the prep necessary; the next lines of code generate the graph itself. They're pretty intuitive; see if you can read them yourself.

(If you're using a different text, be sure to change the title by editing the `ggtitle` line of code before running.)

```
xaxis <- 1:length(word_vector)
yaxis <- rep(0,length(xaxis))
yaxis[keyword] <- 1
word_plot <- data.frame(xaxis, yaxis)

ggplot(data=word_plot, aes(x=xaxis, y=yaxis)) +
  geom_line() +
  labs(x="Narrative Time (in words)") +
  theme(axis.title.y=element_blank(),
        axis.text.y=element_blank(),
        axis.ticks.y=element_blank()) +
  ggtitle("Mentions of Bras Coupe in Cable's The Grandissimes")
```

[3] If this code doesn't execute properly, check to make sure that the name here is spelled c-o-u-p-[e with an accent]; older computers/software sometimes corrupt this last character.



Feel free to try out a few other words if you'd like; all it takes is re-defining the keyword object. See if you can recall how to do that on your own – and if not, just scroll up two code blocks.

It's often useful to know not just where a particular word occurs in the text but also what words appear immediately alongside it. We call these collocate words or collocates.

Let's take this step by step. How would we find the five collocates on either side of just the first instance of our keyword? Since our keyword object in memory is simply a vector of positions, this is simply a matter subtraction and addition.

```
keyword <- which(word_vector=="eyes")

word_vector[(keyword[1]-5):(keyword[1]+5)]
```

```
##  [1] "are"       "beating"   "wit"       "is"        "flashing"
##  [6] "eyes"      "encounter" "eyes"      "with"      "the"
## [11] "leveled"
```

This returns a vector of the ten words surrounding our key word (plus our key word).[4] Well enough! But we want to do this once for every element in keyword. In order to do this, we'll have to use a `for` *loop*. A `for` loop executes a section of code once `for` every element in a vector. We know our vector - keyword - so we'll declare what's called a *loop control variable* with the syntax `(j in seq_along(keyword))`. The code being looped gets put inside curly brackets (`{ }`).

Instead of `keyword[1]`, we index the loop control variable `j`. The first time the loop runs `j` will be 1; the second time it'll be 2; and so on. But wait: how do we save our results? We know that the first line of code

[4]The words here seem a bit difficult to parse, but this is simply because we lack the punctuation to immediately make sense of Cable's syntax: "The dance goes on; hearts are beating, wit is flashing, eyes encounter eyes with the leveled lances of their beams, merriment and joy and sudden bright surprises thrill the breast, voices are throwing off disguise, and beauty's coy ear is bending with a venturesome docility; here love is baffled, there deceived, yonder takes prisoners and here surrenders."

in the loop will produce a vector for each instance of our key word. How do we save a bunch of vectors? With a new data type: a list. A list is basically a vector of vectors. So before running the loop it's necessary to declare an empty list to store our results in. Double brackets are used to index a whole vector into a list. Since we want the results of the loop's first run to be first, those of its second run to be second, and so on, we need to index with the loop control variable once again.

```
collocates <- list()

for (j in seq_along(keyword)) {
  collocates[[j]] <- word_vector[(keyword[j]-5):(keyword[j]+5)]
}
```

Once the loop has finished, type "collocates" into the console; this is what a list looks like. There are plenty of reasons to keep data in this format, but to find the most frequently occurring collocates it is necessary to unlist them.

```
all_collocates <- unlist(collocates)

sort(table(all_collocates), decreasing=TRUE)[1:15]
```

```
## all_collocates
##  eyes   the   her   and   his    of  with    to     a   she    he    in 
##   192   109    97    96    81    61    45    35    27    25    20    19 
##     s again   but 
##    18    15    14 
```

Again, we'll want to try to clean up those results with our stoplist:

```
top_collocates <- all_collocates[-which(all_collocates %in% stoplist)]

sort(table(top_collocates), decreasing=TRUE)[1:25]
```

```
## top_collocates
##       eyes          a          s     lifted frowenfeld     closed 
##        192         27         18         14         12         10 
##     aurora     turned   clotilde      large     opened apothecary 
##          8          8          7          7          7          6 
##      black       cast    dropped          i       back       fell 
##          6          6          6          6          5          5 
##     honoré    instant       blue       face   flashing     glance 
##          5          5          4          4          4          4 
##       head 
##          4 
```

This indicates that there are a few things that should be added to our stoplist. Easily done: just redefine stoplist as a vector that now contains all of stoplist plus two new elements, "a" and "s".[5]

```
stoplist <- c(stoplist, "a", "s")
```

There are other ways to measure context and relationships between words as well, most notably topic modeling tools like `mallet` or `wordVectors`. These, however, become most useful when working with a corpus of texts, even if only a small one.

[5] What if we came to regret a particular addition to our stoplist and wanted to remove it? The surest way is to first make a vector of the character strings you wish to remove and then index stoplist to exclude (`!`) all elements of stoplist that are also `%in%` the remove words vector.

```
remove_words <- c("a", "s")
stoplist <- stoplist[! stoplist %in% remove_words]
```

Sentiment Analysis

We're using the `tidyword` package, which provides a few datasets for this purpose. For starters, type "sentiments" into the console.

```
sentiments
```

```
## # A tibble: 27,314 x 4
##    word        sentiment lexicon score
##    <chr>       <chr>     <chr>   <int>
##  1 abacus      trust     nrc        NA
##  2 abandon     fear      nrc        NA
##  3 abandon     negative  nrc        NA
##  4 abandon     sadness   nrc        NA
##  5 abandoned   anger     nrc        NA
##  6 abandoned   fear      nrc        NA
##  7 abandoned   negative  nrc        NA
##  8 abandoned   sadness   nrc        NA
##  9 abandonment anger     nrc        NA
## 10 abandonment fear      nrc        NA
## # ... with 27,304 more rows
```

Each observation - or row - here corresponds to a single word, as we can see from the word variable - or column - on the left. Next, note the third variable, lexicon. The sentiments dataset actually contains three different sentiment lexicons. The first, "nrc," associates words with a particular affect or emotion; as you can see in the sentiment variable, a word can be associated with multiple sentiments in the "nrc" lexicon. The "bing" lexicon, by contrast, assigns each word either "positive" or "negative" sentiment. Finally, the "afinn" lexicon scores each word between -5 and 5.

Using the "nrc" lexicon, we can identify all the words in our text that are associated with a particular sentiment. I've chosen fear, but you can pick whichever sentiment you'd like: if you do, make sure to change all the instances of the word "fear" in the next couple blocks of code (you can use ctrl+f to find them).

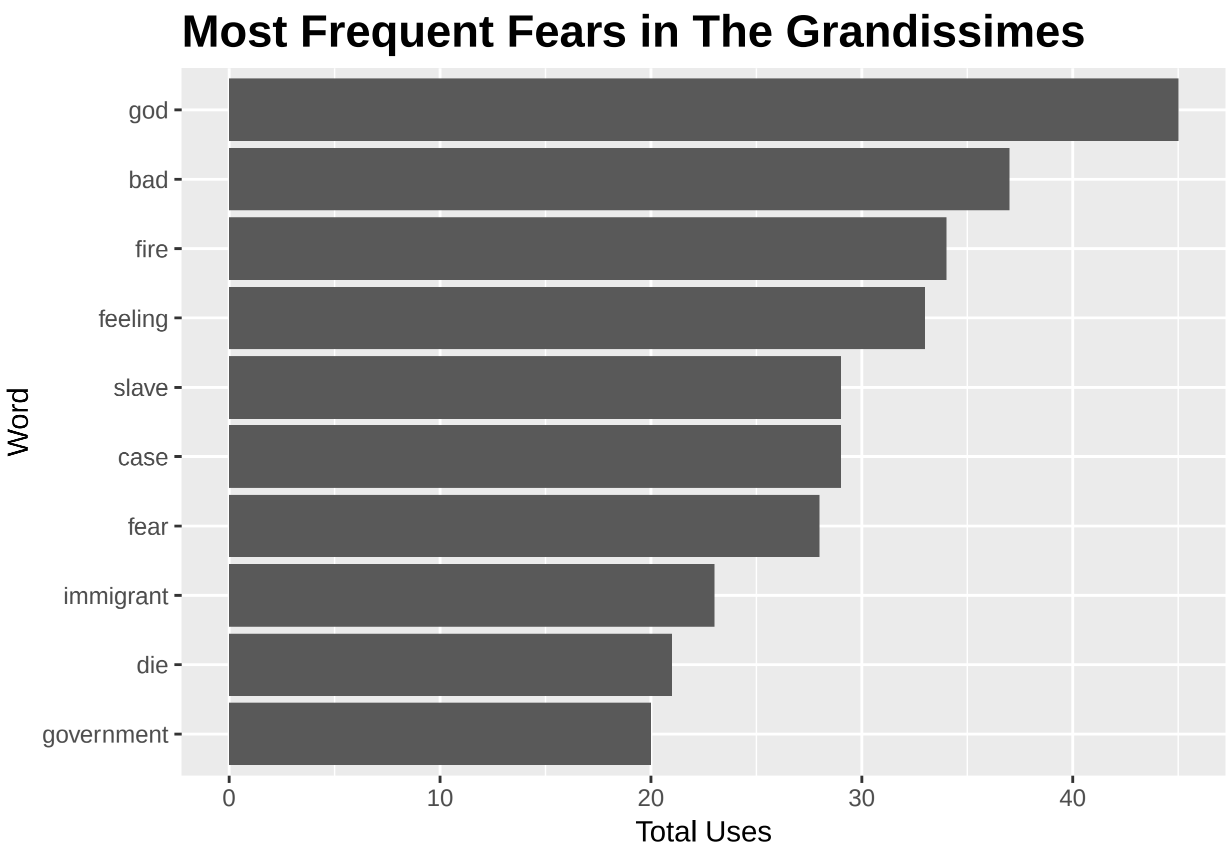
```
fear <- get_sentiments("nrc") %>%
  filter(sentiment=="fear")

fear_words <- word_vector[which(word_vector %in% fear$word)]
sort(table(fear_words), decreasing=TRUE)[1:20]
```

```
## fear_words
##        god        bad       fire    feeling       case      slave
##         45         37         34         33         29         29
##       fear  immigrant        die government      grave      fever
##         28         23         21         20         20         19
##        ill     broken      death      worse   distress      doubt
##         19         18         17         17         16         16
##      wound      broke
##         16         15
```

Part of the utility of using R is that we can easily translate this into graph form. To do so we'll again use the piping operator `%>%` to turn our fear_words vector into a data frame and once again `summarize` the total number of observations for each word.

```
fear_df <- data.frame(fear_words) %>%
  group_by(fear_words) %>%
  summarize(total=n()) %>%
  arrange(desc(total))
```

```r
ggplot(data=fear_df[1:10,], aes(x=reorder(fear_words, total), y=total)) +
  geom_bar(stat="identity") +
  coord_flip() +
  ggtitle("Most Frequent Fears in The Grandissimes") +
  theme(plot.title = element_text(face="bold", size=rel(1.5))) +
  labs(x="Word", y="Total Uses")
```



Now, this chart takes as its basis the entire text. But one might rather want to know the sentiment of one part of a text or compare how it shifts from one part of the text to the next. There are several metrics by which one could do this: using the gutenberg_text object (it's still in our environment!) we could break the text into groups of words by sentence, paragraph, or even chapter.[6]

We'll begin by transforming our word_vector into a dataframe again, with two variables: one for the words and one for their place in the sequence of all words. This time, we'll use the "bing" lexicon to just track whether a word is positive or negative.

```r
word_df <- data.frame(word=word_vector, word_position=seq_along(word_vector))

text_positivity <- inner_join(word_df, get_sentiments("bing"), by="word")
```

This next bit of code relies on a number of new functions, so don't worry if it just seems hazy. First we

[6] Recall our initial decision to split up the text into words - that is, by non-letter characters - looked like this:

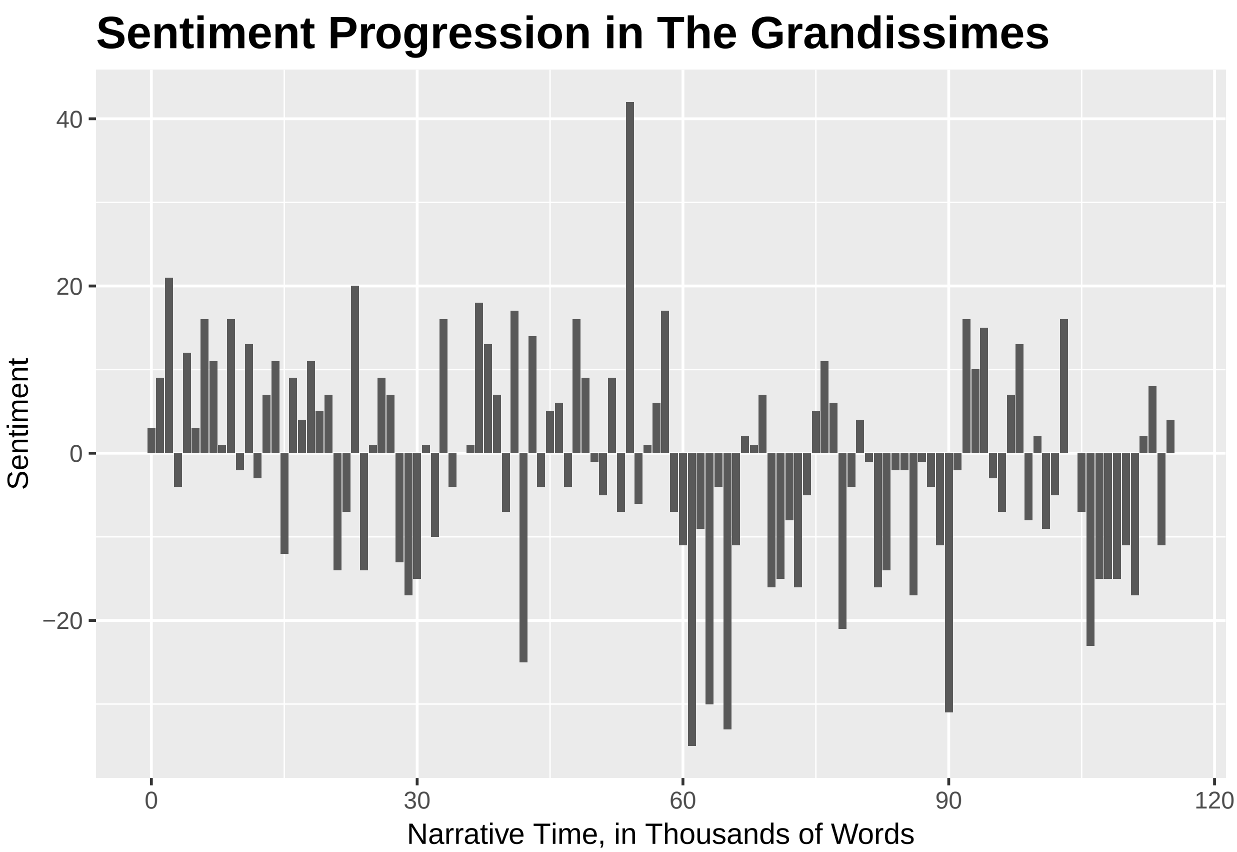
```r
words <- unlist(strsplit(text, "\\W+"))
```

If we wanted to instead split up the text into sentences - that is, by sentence-ending punctuation - we would write something like this, using vertical lines (|) to designate a split at either a literal period, question mark, or exclamation mark:

```r
sentences <- unlist(strsplit(text, "\\.|\\?|\\!"))
```

count the number of observations of each sentiment in each group of 1000 words in our original word_vector, which reorganizes the data frame so that each observation is one sentiment from one group rather than one individual word. Then we spread the sentiment variable so that each of its two possible values, positive and negative, is its own variable whose value is the total number (n) of words associated with it in that 1000-word group. Finally, we mutate to add a new variable, sentiment_quant, which simply subtracts the difference between the value in the positive variable and the value in the negative variable.

```
sentiment_plot <- text_positivity %>%
  count(index = word_position %/% 1000, sentiment) %>%
  spread(sentiment, n, fill = 0) %>%
  mutate(sentiment_quant = positive - negative)

sentiment_plot %>%
  ggplot(aes(x=index, y=sentiment_quant)) +
  geom_col() +
  ggtitle("Sentiment Progression in The Grandissimes") +
  theme(plot.title = element_text(face="bold", size=rel(1.5))) +
  labs(x="Narrative Time, in Thousands of Words", y="Sentiment")
```



Now this gives us a lot to chew over. In the case of *The Grandissimes*, we have a text that is unusually gloomy for an unusually large amount of the second half of the text despite being generally chipper for the first half. The novel's resolution barely gets into positive territory, and only at the last minute. This makes *The Grandissimes* somewhat unusual, despite the fact that it does indeed possess love plots and resolution of conflict – formal attributes that might lead one to mistakenly consider the novel simply another example of the postbellum historical romance.

Of course, sentiment analysis is only as useful as the lexicon being used, and a lexicon that may be particularly useful for one text might not be for another text. To check this, we might want to do a bit of spot checking to see that sentiments are being assigned in a way that we deem appropriate. We might also want to compare different sentiment lexicons: for example, we could make a second version of this same graph that instead uses the "afinn" lexicon and compare the results. This kind of self-review is essential when working with quantitative methods.

There's a bigger moral here too: that quantitative analysis should always be in dialogue with other qualitative forms of analysis. The benefits of quantitative analysis lie in its ability to articulate aspects of texts that can often be more precise or succinct than qualitative analysis and to direct our attention to further areas for qualitative analysis that we might've overlooked.

Finally, we would really appreciate it if you took a minute to fill out our brief feedback survey.

If you'd like to look at this workshop in more detail or run the code yourself, visit https://github.com/azleslie/TextAnalysisIntro. If you want to try it out on a different text, simply save that text as a .txt file and save it into the sample_texts directory.

Thanks for participating!