Code ▾

Basics of Text Mining in R - Bag of Words

Introduction

So what is text mining? To put it simple: Text mining is the process of distilling actionable insights from text. In this article we'll be dealing with the so called Bag of Words, i.e. BoW approach to text mining.

I'm a big fan of first do than talk about approach in learning so let's jump right into easy practical examples and build the story of text mining from there.

Get a bit of taste of text mining: `qdap` and counting terms

At its heart, **bag of words** text mining represents a way to count terms, or **n-grams**, across a collection of documents. Consider the following sentences, which we've saved to `text` and made available in the workspace:

Hide

```
text <- "Text mining usually involves the process of structuring the input text. The overarch
ing goal is, essentially, to turn text into data for analysis, via application of natural lan
guage processing (NLP) and analytical methods."
```

Manually counting words in the sentences above is a pain! Fortunately, the `qdap` package offers a better alternative. You can easily find the top 3 most frequent terms (including ties) in text by calling the `freq_terms` function and specifying 3.

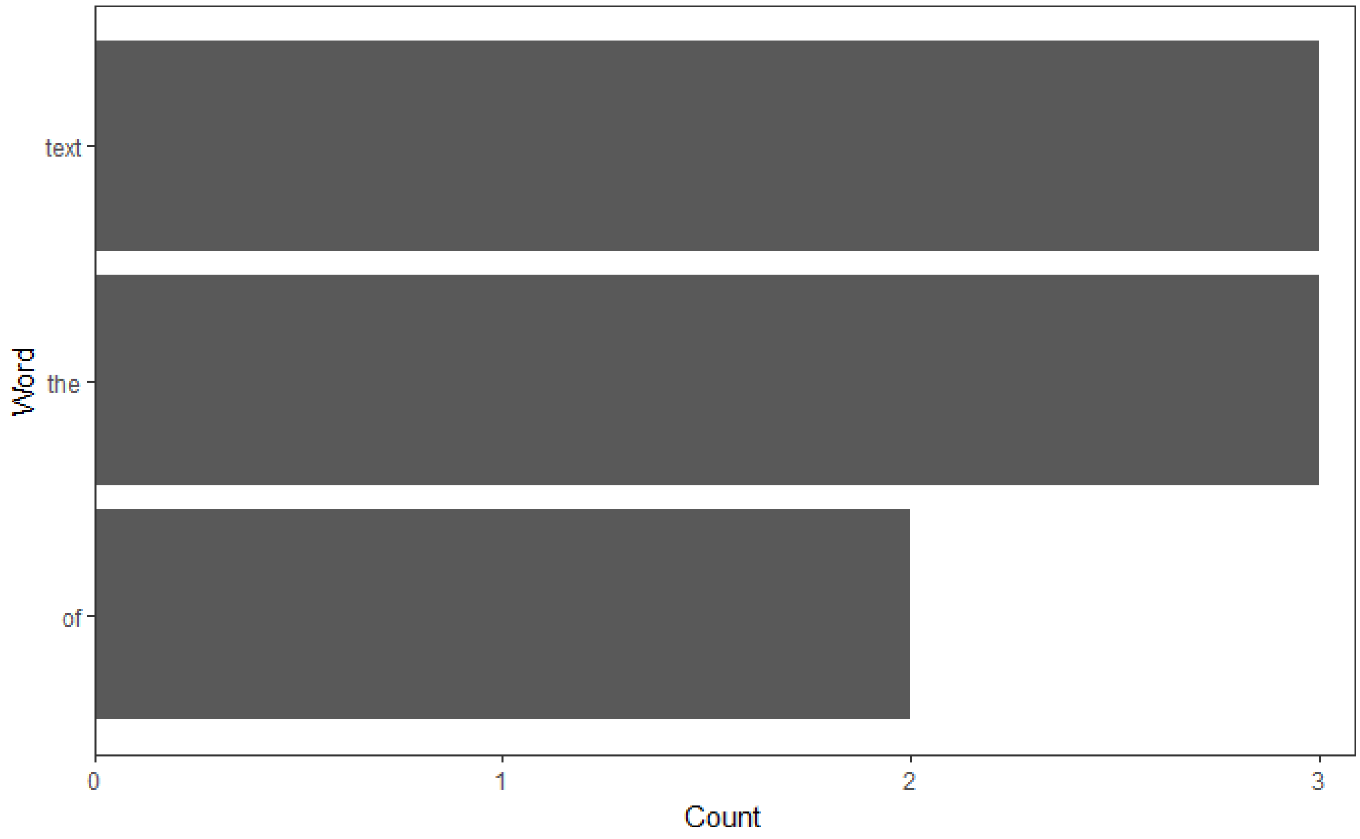
Hide

```
library(qdap)
frequent_terms <- freq_terms(text, 3)
```

The `frequent_terms` object stores all unique words and their counts. You can then make a bar chart simply by calling the plot function on the `frequent_terms` object.

Hide

```
plot(frequent_terms)
```



From loading the textual data to TDM and DTM: short examples

The first step of text mining endeavour is of course loading the very textual data that is supposed to be analyzed.

Hide

```
library(readr)
```

```
package 拹粭readr拹牸 was built under R version 3.3.2
```

Hide

```
# Import text data
tweets <- read_csv("data/NeildeGrasseTysonTweets.csv")
```

```
Missing column names filled in: 'X1' [1]Parsed with column specification:
cols(
  X1 = col_integer(),
  date = col_character(),
  id = col_double(),
  link = col_character(),
  retweet = col_character(),
  text = col_character(),
  author = col_character()
)
```

Hide

```
# View the structure of tweets
str(tweets)
```

```
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':   2428 obs. of  7 variables:
 $ X1     : int  0 1 2 3 4 5 6 7 8 9 ...
 $ date   : chr  "Aug 21" "Oct 9" "Oct 9" "Oct 7" ...
 $ id     : num  7.67e+17 7.85e+17 7.85e+17 7.84e+17 7.84e+17 ...
 $ link   : chr  "/neiltyson/status/767371694834978817" "/neiltyson/status/78518663694663680
0" "/neiltyson/status/785131023923314688" "/neiltyson/status/784443331568930817" ...
 $ retweet: chr  "False" "False" "False" "False" ...
 $ text   : chr  "Moon’s shadow landfalls Oregon, crosses USA at 1800mph, exits SCarolina. Be
hold ‘Muuurica’s Eclipse.pic.twitter.com/fIMCnEyyQy""| __truncated__ "@huggy_panda  Oink, oin
k.   : - )" "Future headlines from the Multiverse: Nov 9, 2016: “Trump: How I Got Hillary Ele
cted while Dismantling the Republican Party.”""| __truncated__ "Awww. That’s the nicest thing
anybody has said to me in a long while.https://twitter.com/ayeshatron/status/7844414326523207
69 …"| __truncated__ ...
 $ author : chr  "deGrasseTyson" "deGrasseTyson" "deGrasseTyson" "deGrasseTyson" ...
 - attr(*, "spec")=List of 2
  ..$ cols   :List of 7
  .. ..$ X1     : list()
  .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
  .. ..$ date   : list()
  .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
  .. ..$ id     : list()
  .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
  .. ..$ link   : list()
  .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
  .. ..$ retweet: list()
  .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
  .. ..$ text   : list()
  .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
  .. ..$ author : list()
  .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
  ..$ default: list()
  .. ..- attr(*, "class")= chr  "collector_guess" "collector"
  ..- attr(*, "class")= chr "col_spec"
```

Hide

```
# Print out the number of rows in tweets
nrow(tweets)
```

```
[1] 2428
```

Hide

```
# Isolate text from tweets: tweets_text
tweets_text <- tweets$text
str(tweets_text)
```

```
 chr [1:2428] "Moon’s shadow landfalls Oregon, crosses USA at 1800mph, exits SCarolina. Behol
d ‘Muuurica’s Eclipse.pic.twitter.com/fIMCnEyyQy""| __truncated__ ...
```

Building a corpus

Let’s now build a corpus out of this vector of strings. A corpus is a collection of documents, but it’s also important to know that in the `tm` domain, R recognizes it as a separate data type.

There are two kinds of the corpus data type, the permanent corpus, i.e. PCorpus, and the volatile corpus, i.e. VCorpus. In essence, the difference between the two has to do with how the collection of documents is stored in your computer. We will use the volatile corpus, which is held in computer’s RAM rather than saved to disk, just to be more memory efficient.

To make a volatile corpus, R needs to interpret each element in our vector of text, `tweets_text` , as a document. And the `tm` package provides what are called Source functions to do just that! In this exercise, we’ll use a Source function called `VectorSource()` because our text data is contained in a vector. The output of this function is called a *Source object*.

Hide

```
library(tm)
tweets_source <- VectorSource(tweets_text)
```

Now that we’ve converted our vector to a Source object, we pass it to another `tm` function, `VCorpus()` , to create our volatile corpus. The `VCorpus` object is a nested list, or list of lists. At each index of the `VCorpus` object, there is a `PlainTextDocument` object, which is essentially a list that contains the actual text data (`content`), as well as some corresponding metadata (`meta`) which can help to visualize a `VCorpus` object and to conceptualize the whole thing.

Hide

```
# Make a volatile corpus: tweets_corpus
tweets_corpus <- VCorpus(tweets_source)
# Print out the tweets_corpus
tweets_corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2428
```

Hide

```
# Print data on the 15th tweet in tweets_corpus
tweets_corpus[[15]]
```

```
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 117
```

Hide

```
# Print the content of the 15th tweet in tweets_corpus
tweets_corpus[[15]][1]
```

```
$content
[1] "There’s more that 300 metric tons of it embedded in every 500-meter metallic asteroid th
at orbits the Sun. #ThatsGold"
```

Hide

```
str(tweets_corpus[[15]])
```

```
List of 2
 $ content: chr "There’s more that 300 metric tons of it embedded in every 500-meter metallic
asteroid that orbits the Sun. #ThatsGold""| __truncated__
 $ meta   :List of 7
  ..$ author       : chr(0)
  ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:13"
  ..$ description  : chr(0)
  ..$ heading      : chr(0)
  ..$ id           : chr "15"
  ..$ language     : chr "en"
  ..$ origin       : chr(0)
  ..- attr(*, "class")= chr "TextDocumentMeta"
 - attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
```

Because another common text source is a data frame, there is a Source function called `DataframeSource()`. The `DataframeSource()` function treats the entire row as a complete document, so be careful not to pick up non-text data like customer IDs when sourcing a document this way.

Hide

```
example_text <- data.frame(num = c(1,2,3), Author1 = c("Text mining is a great time.", "Text
 analysis provides insights", "qdap and tm are used in text mining"), Author2 = c("R is a gre
at language", "R has many uses", "R is cool!"), stringsAsFactors = FALSE)
# Create a DataframeSource on columns 2 and 3: df_source
df_source <- DataframeSource(example_text[, 2:3])
# Convert df_source to a corpus: df_corpus
df_corpus <- VCorpus(df_source)
# Examine df_corpus
df_corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 3
```

Hide

```
str(df_corpus)
```

```
List of 3
 $ 1:List of 2
  ..$ content: chr [1:2] "Text mining is a great time." "R is a great language"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:14"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "1"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
 $ 2:List of 2
  ..$ content: chr [1:2] "Text analysis provides insights" "R has many uses"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:14"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "2"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
 $ 3:List of 2
  ..$ content: chr [1:2] "qdap and tm are used in text mining" "R is cool!"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:14"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "3"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
 - attr(*, "class")= chr [1:2] "VCorpus" "Corpus"
```

Hide

```
# Create a VectorSource on column 3: vec_source
vec_source <- VectorSource(example_text[, 3])
# Convert vec_source to a corpus: vec_corpus
vec_corpus <- VCorpus(vec_source)
# Examine vec_corpus
vec_corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 3
```

Hide

```
str(vec_corpus)
```

```
List of 3
 $ 1:List of 2
  ..$ content: chr "R is a great language"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:14"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "1"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
 $ 2:List of 2
  ..$ content: chr "R has many uses"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:14"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "2"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
 $ 3:List of 2
  ..$ content: chr "R is cool!"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-03-07 16:00:14"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "3"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
 - attr(*, "class")= chr [1:2] "VCorpus" "Corpus"
```

Cleaning and preprocessing of the text

After obtaining the corpus, usually, the next step will be cleaning and preprocessing of the text. For this endeavor we are mostly going to use functions from the `tm` and `qdap` packages. In bag of words text mining, cleaning helps aggregate terms. For example, it may make sense that the words “miner”, “mining” and “mine” should be considered one term. Specific preprocessing steps will vary based on the project. For example, the words used in tweets are vastly different than those used in legal documents, so the cleaning process can also be quite different.

Common preprocessing functions include:

- `tolower()` : Make all characters lowercase
- `removePunctuation()` : Remove all punctuation marks
- `removeNumbers()` : Remove numbers
- `stripWhitespace()` : Remove excess whitespace

Note that `tolower()` is part of base `R` , while the other three functions come from the `tm` package.

Let’s check how this functions work on a small chunk of plain text:

Hide

```
# Create the object: text
text <- "<b>She</b> woke up at       6 A.M. It\'s so early!  She was only 10% awake and began
 drinking coffee in front of her computer."
# All lowercase
tolower(text)
```

```
[1] "<b>she</b> woke up at       6 a.m. it's so early!  she was only 10% awake and began drin
king coffee in front of her computer."
```

Hide

```
# Remove punctuation
removePunctuation(text)
```

```
[1] "bSheb woke up at       6 AM Its so early  She was only 10 awake and began drinking coffe
e in front of her computer"
```

Hide

```
# Remove numbers
removeNumbers(text)
```

```
[1] "<b>She</b> woke up at        A.M. It's so early!  She was only % awake and began drinkin
g coffee in front of her computer."
```

Hide

```
# Remove whitespace
stripWhitespace(text)
```

```
[1] "<b>She</b> woke up at 6 A.M. It's so early! She was only 10% awake and began drinking co
ffee in front of her computer."
```

The `qdap` package offers other text cleaning functions. Each is useful in its own way and is particularly powerful when combined with the others.

- `bracketX()` : Remove all text within brackets (e.g. “It’s (so) cool” becomes “It’s cool”)
- `replace_number()` : Replace numbers with their word equivalents (e.g. “2” becomes “two”)
- `replace_abbreviation()` : Replace abbreviations with their full text equivalents (e.g. “Sr” becomes “Senior”)
- `replace_contraction()` : Convert contractions back to their base words (e.g. “shouldn’t” becomes “should not”)
- `replace_symbol()` Replace common symbols with their word equivalents (e.g. “$” becomes “dollar”)

Let’s try out some of these functions on the `text` string we’ve defined in the previous example:

Hide

```
# Remove text within brackets
bracketX(text)
```

```
[1] "She woke up at 6 A.M. It's so early! She was only 10% awake and began drinking coffee in
front of her computer."
```

Hide

```
# Replace numbers with words
replace_number(text)
```

```
[1] "<b>She</b> woke up at six A.M. It's so early! She was only ten% awake and began drinking
coffee in front of her computer."
```

Hide

```
# Replace abbreviations
replace_abbreviation(text)
```

```
[1] "<b>She</b> woke up at 6 AM It's so early! She was only 10% awake and began drinking coff
ee in front of her computer."
```

Hide

```
# Replace contractions
replace_contraction(text)
```

```
[1] "<b>She</b> woke up at 6 A.M. it is so early! She was only 10% awake and began drinking c
offee in front of her computer."
```

Hide

```
# Replace symbols with words
replace_symbol(text)
```

```
[1] "<b>She</b> woke up at 6 A.M. It's so early! She was only 10 percent awake and began drin
king coffee in front of her computer."
```

Stop words

The next issue that we'll deal with are the so-called *stop words*. These the are words that are frequent but provide little information. So you may want to remove them. Some common English stop words include "I", "she'll", "the", etc. In the `tm` package, there are 174 stop words on this common list. In fact, when you are doing an analysis you will likely need to add to this list. Leaving certain frequent words that don't add any insight will cause them to be overemphasized in a frequency analysis which usually leads to wrongly biased interpretation of results.

Using the `c()` function allows you to add new words (separated by commas) to the stop words list. For example, the following would add "word1" and "word2" to the default list of English stop words:

```
all_stops <- c("word1", "word2", stopwords("en"))
```

Once you have a list of stop words that makes sense, you will use the `removeWords()` function on your text. `removeWords()` takes two arguments: the text object to which it's being applied and the list of words to remove.

Hide

```
# List standard English stop words
stopwords("en")
```

```
  [1] "i"          "me"         "my"         "myself"     "we"         "our"        "ours"      
    "ourselves" 
  [9] "you"        "your"       "yours"      "yourself"   "yourselves" "he"         "him"       
    "his"       
 [17] "himself"    "she"        "her"        "hers"       "herself"    "it"         "its"       
    "itself"    
 [25] "they"       "them"       "their"      "theirs"     "themselves" "what"       "which"     
    "who"       
 [33] "whom"       "this"       "that"       "these"      "those"      "am"         "is"        
    "are"       
 [41] "was"        "were"       "be"         "been"       "being"      "have"       "has"       
    "had"       
 [49] "having"     "do"         "does"       "did"        "doing"      "would"      "should"    
    "could"     
 [57] "ought"      "i'm"        "you're"     "he's"       "she's"      "it's"       "we're"     
    "they're"   
 [65] "i've"       "you've"     "we've"      "they've"    "i'd"        "you'd"      "he'd"      
    "she'd"     
 [73] "we'd"       "they'd"     "i'll"       "you'll"     "he'll"      "she'll"     "we'll"     
    "they'll"   
 [81] "isn't"      "aren't"     "wasn't"     "weren't"    "hasn't"     "haven't"    "hadn't"    
    "doesn't"   
 [89] "don't"      "didn't"     "won't"      "wouldn't"   "shan't"     "shouldn't"  "can't"     
    "cannot"    
 [97] "couldn't"   "mustn't"    "let's"      "that's"     "who's"      "what's"     "here's"    
    "there's"   
[105] "when's"     "where's"    "why's"      "how's"      "a"          "an"         "the"       
    "and"       
[113] "but"        "if"         "or"         "because"    "as"         "until"      "while"     
    "of"        
[121] "at"         "by"         "for"        "with"       "about"      "against"    "between"   
    "into"      
[129] "through"    "during"     "before"     "after"      "above"      "below"      "to"        
    "from"      
[137] "up"         "down"       "in"         "out"        "on"         "off"        "over"      
    "under"     
[145] "again"      "further"    "then"       "once"       "here"       "there"      "when"      
    "where"     
[153] "why"        "how"        "all"        "any"        "both"       "each"       "few"       
    "more"      
[161] "most"       "other"      "some"       "such"       "no"         "nor"        "not"       
    "only"      
[169] "own"        "same"       "so"         "than"       "too"        "very"      
```

Hide

```
# Print text without standard stop words
removeWords(text, stopwords("en"))
```

```
[1] "<b>She</b> woke         6 A.M. It's  early!  She   10% awake  began drinking coffee  fro
nt   computer."
```

Hide

```
# Add "coffee" and "bean" to the list: new_stops
new_stops <- c("coffee", "bean", stopwords("en"))
# Remove stop words from text
removeWords(text, new_stops)
```

```
[1] "<b>She</b> woke         6 A.M. It's  early!  She   10% awake  began drinking   front   c
omputer."
```

Intro to word stemming and stem completion

Still another useful preprocessing step involves word stemming and stem completion. The tm package provides the stemDocument() function to get to a word's root. This function either takes in a character vector and returns a character vector, or takes in a PlainTextDocument and returns a PlainTextDocument.

Still another useful preprocessing step involves *word stemming* and *stem completion*. The `tm` package provides the `stemDocument()` function to get to a word's root. This function either takes in a character vector and returns a character vector, or takes in a `PlainTextDocument` and returns a `PlainTextDocument`. For example,

```
stemDocument(c("computational", "computers", "computation"))
```

returns "comput" "comput" "comput". But because "comput" isn't a real word, we want to re-complete the words so that "computational", "computers", and "computation" all refer to the same word, say "computer", in our ongoing analysis.

We can easily do this with the `stemCompletion()` function, which takes in a character vector and an argument for the completion dictionary. The completion dictionary can be a character vector or a Corpus object. Either way, the completion dictionary for our example would need to contain the word "computer" for all the words to refer to it.

Hide

```
# Create complicate
complicate <- c("complicated", "complication", "complicatedly")
# Perform word stemming: stem_doc
stem_doc <- stemDocument(complicate)
# Create the completion dictionary: comp_dict
comp_dict <- ("complicate")
# Perform stem completion: complete_text
complete_text <- stemCompletion(stem_doc, comp_dict)
# Print complete_text
complete_text
```

```
     complic      complic      complic 
"complicate" "complicate" "complicate" 
```

Word stemming and stem completion on a sentence

Let’s consider the following sentence as our document for this exercise:

> “In a complicated haste, Tom rushed to fix a new complication, too complicatedly.”

This sentence contains the same three forms of the word “complicate” that we saw in the previous exercise. The difference here is that even if you called `stemDocument()` on this sentence, it would return the sentence without stemming any words.

Hide

```
stemDocument("In a complicated haste, Tom rushed to fix a new complication, too complicatedl
y.")
```

```
[1] "In a complicated haste, Tom rushed to fix a new complication, too complicatedly."
```

This happens because `stemDocument()` treats the whole sentence as one word. In other words, our document is a character vector of length 1, instead of length n, where n is the number of words in the document. To solve this problem, we first remove the punctuation marks with the `removePunctuation()` function, we then `strsplit()` this character vector of length 1 to length n, `unlist()` , then proceed to stem and re-complete.

Hide

```
text_data <- "In a complicated haste, Tom rushed to fix a new complication, too complicatedl
y."
# Remove punctuation: rm_punc
rm_punc <- removePunctuation(text_data)
# Create character vector: n_char_vec
n_char_vec <- unlist(strsplit(rm_punc, split = ' '))
# Perform word stemming: stem_doc
stem_doc <- stemDocument(n_char_vec)
# Print stem_doc
stem_doc
```

```
 [1] "In"      "a"       "complic" "hast"    "Tom"     "rush"    "to"      "fix"     "a"
  "new"
[11] "complic" "too"     "complic"
```

Hide

```
# Create the completion dictionary: comp_dict
comp_dict <- c("In", "a", "complicate", "haste", "Tom", "rush", "to", "fix", "new", "too")
# Re-complete stemmed document: complete_doc
complete_doc <- stemCompletion(stem_doc, comp_dict)
# Print complete_doc
complete_doc
```

```
          In            a       complic          hast          Tom         rush            to 
       fix 
        "In"          "a" "complicate"       "haste"        "Tom"       "rush"          "to" 
     "fix" 
           a          new       complic           too       complic 
         "a"        "new" "complicate"         "too" "complicate" 
```

Applying preprocessing steps to a corpus

The `tm` package provides a special function `tm_map()` to apply cleaning functions to a corpus. Mapping these functions to an entire corpus makes scaling the cleaning steps very easy.

To save time (and lines of code) it's a good idea to use a custom function, since you may be applying the same functions over multiple corpora. You can probably guess what the `clean_corpus()` function does. It takes one argument, corpus, and applies a series of cleaning functions to it in order, then returns the final result.

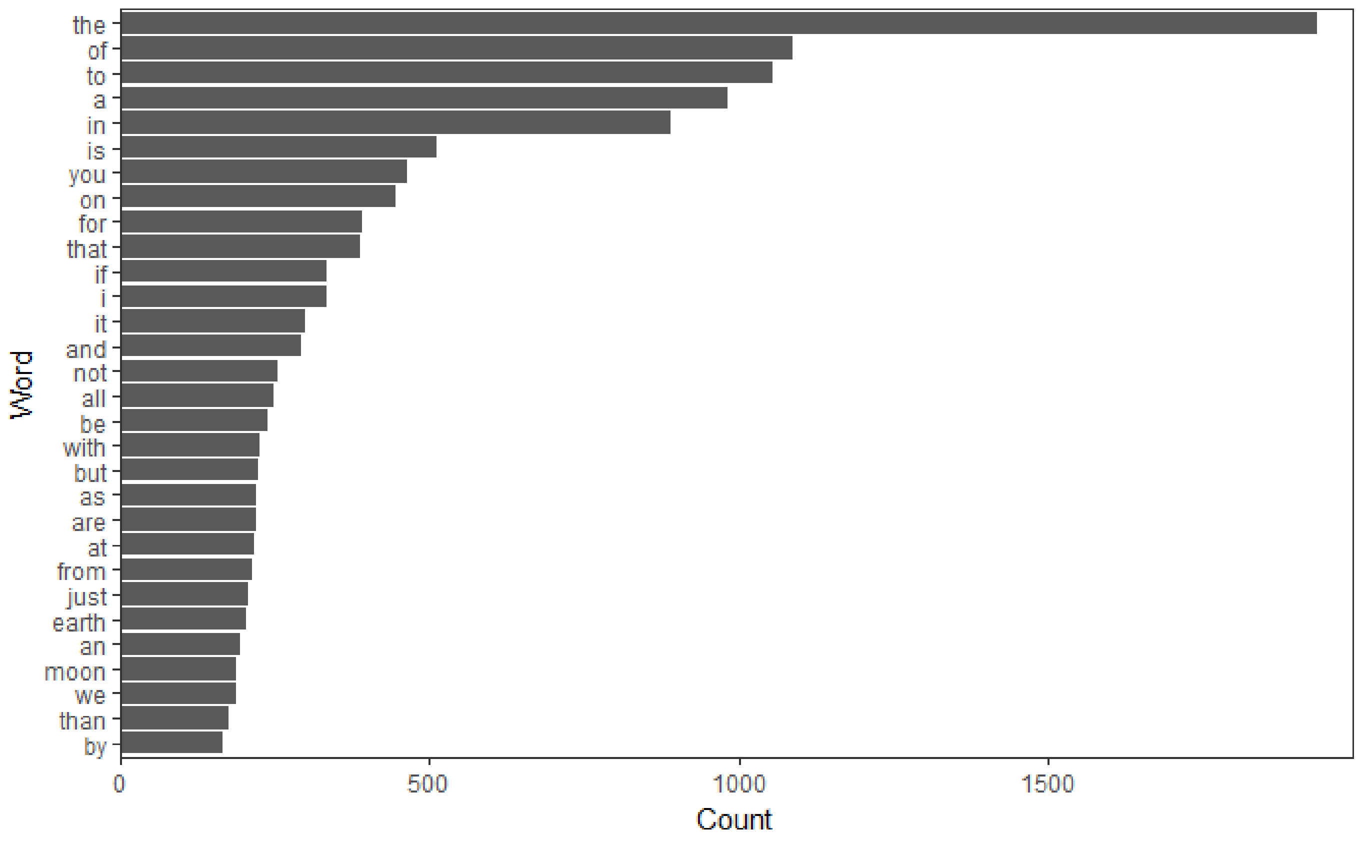
Notice how the `tm` package functions do not need `content_transformer()`, but base `R` and `qdap` functions do.

Be sure to test your function's results. If you want to draw out currency amounts, then `removeNumbers()` shouldn't be used! Plus, the order of cleaning steps makes a difference. For example, if you `removeNumbers()` and then `replace_number()`, the second function won't find anything to change!

Check, check, and re-check!

Hide

```
#Let's find the most frequent words in our tweets_text and see whether we should get rid of s
ome
frequent_terms <- freq_terms(tweets_text, 30)
plot(frequent_terms)
```



Hide

```
# Well nothing stands out in particular, exepct ties and articles, so the standard wocabulary
 of stopwords
# in English will do just fine.
# Create the custom function that will be used to clean the corpus: clean_coupus
clean_corpus <- function(corpus){
  corpus <- tm_map(corpus, stripWhitespace)
  corpus <- tm_map(corpus, removePunctuation)
  corpus <- tm_map(corpus, content_transformer(tolower))
  corpus <- tm_map(corpus, removeWords, stopwords("en"))
    return(corpus)
}
# Apply your customized function to the tweet_corp: clean_corp
clean_corp <- clean_corpus(tweets_corpus)
# Print out a cleaned up tweet
clean_corp[[227]][1]
```

```
$content
[1] "thanks   birthday wellwishers   twitterverse born  1958 yet  don’t feel  day  57"
```

Hide

```
# Print out the same tweet in original form
tweets$text[227]
```

```
[1] "Thanks to all Birthday well-wishers in the Twitterverse.  Born in 1958, yet I don’t feel
a day over 57."
```

Making a document-term matrix

The document-term matrix is used when you want to have each document represented as a row. This can be useful if you are comparing authors within rows, or the data is arranged chronologically and you want to preserve the time series.

Hide

```
# Create the dtm from the corpus:
tweets_dtm <- DocumentTermMatrix(clean_corp)
# Print out tweets_dtm data
tweets_dtm
```

```
<<DocumentTermMatrix (documents: 2428, terms: 8406)>>
Non-/sparse entries: 26328/20383440
Sparsity           : 100%
Maximal term length: 107
Weighting          : term frequency (tf)
```

Hide

```
# Convert tweets_dtm to a matrix: tweets_m
tweets_m <- as.matrix(tweets_dtm)
# Print the dimensions of tweets_m
dim(tweets_m)
```

```
[1] 2428 8406
```

Hide

```
# Review a portion of the matrix
tweets_m[148:150, 2587:2590]
```

```
     Terms
Docs  dying ear earlier earliest
  148     0   0       0        0
  149     0   0       0        0
  150     0   0       0        0
```

Hide

```
# Since the sparsity is so high, i.e. a proportion of cells with 0s/ cells with other values
 is too large,
# let's remove some of these low frequency terms
tweets_dtm_rm_sparse <- removeSparseTerms(tweets_dtm, 0.98)
# Print out tweets_dtm data
tweets_dtm_rm_sparse
```

```
<<DocumentTermMatrix (documents: 2428, terms: 40)>>
Non-/sparse entries: 3258/93862
Sparsity           : 97%
Maximal term length: 8
Weighting          : term frequency (tf)
```

Hide

```
# Convert tweets_dtm to a matrix: tweets_m
tweets_m <- as.matrix(tweets_dtm_rm_sparse)
# Print the dimensions of tweets_m
dim(tweets_m)
```

```
[1] 2428   40
```

Hide

```
# Review a portion of the matrix
tweets_m[148:158, 10:22]
```

```
     Terms
Docs  full fyi get good happy just know life like moon never new night
  148    0   1   0    0     0    1    0    0    0    0     0   0     0
  149    0   0   0    0     0    0    0    0    0    1     0   0     0
  150    0   0   0    0     0    0    0    0    0    0     0   0     0
  151    0   0   0    0     0    0    0    0    0    0     0   0     0
  152    0   1   0    0     1    0    0    0    0    0     0   1     0
  153    0   0   0    0     0    0    1    0    0    0     0   0     0
  154    0   0   0    0     0    0    0    0    0    0     0   0     0
  155    1   0   0    0     0    0    0    0    0    1     0   0     0
  156    0   1   0    0     0    0    0    0    0    0     0   0     0
  157    0   0   0    0     0    0    0    0    0    0     0   0     0
  158    0   0   0    0     0    0    0    0    0    0     0   0     0
```

Making a term-document matrix

The TDM is often the matrix used for language analysis. This is because you likely have more terms than authors or documents and life is generally easier when you have more rows than columns. An easy way to start analyzing the information is to change the matrix into a simple matrix using `as.matrix()` on the TDM.

Hide

```
# Create the tdm from the corpus:
tweets_tdm <- TermDocumentMatrix(clean_corp)
# Print out tweets_tdm data
tweets_tdm
```

```
<<TermDocumentMatrix (terms: 8406, documents: 2428)>>
Non-/sparse entries: 26328/20383440
Sparsity           : 100%
Maximal term length: 107
Weighting          : term frequency (tf)
```

Hide

```
# Convert tweets_tdm to a matrix: tweets_m
tweets_m <- as.matrix(tweets_tdm)
# Print the dimensions of tweets_m
dim(tweets_m)
```

```
[1] 8406 2428
```

Hide

```
# Review a portion of the matrix
tweets_m[148:158, 126:138]
```

```
           Docs
Terms       126 127 128 129 130 131 132 133 134 135 136 137 138
  12m         0   0   0   0   0   0   0   0   0   0   0   0   0
  12mid       0   0   0   0   0   0   0   0   0   0   0   0   0
  12th        0   0   0   0   0   0   0   0   0   0   0   0   0
  12x4        0   0   0   0   0   0   0   0   0   0   0   0   0
  130200pm    0   0   0   0   0   0   0   0   0   0   0   0   0
  130mph      0   0   0   0   0   0   0   0   0   0   0   0   0
  132pm       0   0   0   0   0   0   0   0   0   0   0   0   0
  137th       0   0   0   0   0   0   0   0   0   0   0   0   0
  138         0   0   0   0   0   0   0   0   0   0   0   0   0
  1382        0   0   0   0   0   0   0   0   0   0   0   0   0
  13episode   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Hide

```
# Since the sparsity is so high, i.e. a proportion of cells with 0s/ cells with other values
 is too large,
# let's remove some of these low frequency terms
tweets_tdm_rm_sparse <- removeSparseTerms(tweets_tdm, 0.99)
# Print out tweets_dtm data
tweets_tdm_rm_sparse
```

```
<<TermDocumentMatrix (terms: 131, documents: 2428)>>
Non-/sparse entries: 6196/311872
Sparsity           : 98%
Maximal term length: 15
Weighting          : term frequency (tf)
```

Hide

```
# Convert tweets_dtm to a matrix: tweets_m
tweets_m <- as.matrix(tweets_tdm_rm_sparse)
# Print the dimensions of tweets_m
dim(tweets_m)
```

```
[1]  131 2428
```

Hide

```
# Review a portion of the matrix
tweets_m[14:28, 10:22]
```

```
         Docs
Terms      10 11 12 13 14 15 16 17 18 19 20 21 22
  best      0  2  0  0  0  0  0  0  0  0  0  0  0
  big       0  0  0  0  0  0  0  0  0  0  0  0  0
  brain     0  0  0  0  0  0  0  0  0  0  0  0  0
  call      0  0  0  0  0  0  0  0  0  0  0  0  0
  called    0  0  0  0  0  0  0  0  0  0  0  0  0
  can       0  0  0  0  0  0  1  0  0  0  0  0  0
  cool      0  0  0  0  0  0  0  0  0  0  0  0  0
  cosmic    0  0  0  0  0  0  0  0  0  0  0  0  0
  cosmos    0  0  0  0  0  0  0  0  0  0  0  0  0
  curious   0  0  0  0  0  0  0  0  0  0  0  0  0
  day       0  0  1  0  0  0  0  0  0  0  0  0  0
  days      0  0  0  0  0  0  0  0  0  0  0  0  0
  dont      0  0  0  0  0  0  0  0  0  0  0  0  0
  earth     0  0  1  0  0  0  0  0  0  0  0  0  0
  earths    0  0  0  0  0  0  0  0  0  0  0  0  0
```