1 Jumping into text mining

In this chapter, you'll learn the basics of using the bag of words method for analyzing text data.

What is text mining?

Text mining is the process of distilling actionable insights from text.

Quick taste of text mining

It is always fun to jump in with a quick and easy example. Sometimes we can find out the author's intent and main ideas just by looking at the most common words.

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 your workspace:

```
text <- "Text mining usually involves the process of structuring the input text. The overarching goal is, essentially, to turn text into data for analysis, via application of natural language 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 4 most frequent terms (including ties) in `text` by calling the `freq_terms` function and specifying `4`.

```
frequent_terms <- freq_terms(text, 4)
```

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

```
plot(frequent_terms)
```

```
# Load qdap

library(qdap)


# Print new_text to the console

new_text


# Find the 10 most frequent terms: term_count

term_count <- freq_terms(new_text,10)
```

```
# Plot term_count

plot(term_count)
```

Getting started
Load some text

Text mining begins with loading some text data into R, which we'll do with the `read.csv()` function. By default, `read.csv()` treats `character` strings as `factor` levels like `Male`/`Female`. To prevent this from happening, it's very important to use the argument `stringsAsFactors = FALSE`.
A best practice is to examine the object you read in to make sure you know which column(s) are important. The `str()` function provides an efficient way of doing this. You can also count the number of documents using the `nrow()` function on the new object. In this example, it will tell you how many coffee tweets are in the vector.
If the data frame contains columns that are not text, you may want to make a new object using only the correct column of text (e.g. `some_object$column_name`).

```
# Import text data

tweets <- read.csv("coffee.csv", stringsAsFactors = FALSE)


# View the structure of tweets

str(tweets)


# Print out the number of rows in tweets

nrow(tweets)


# Isolate text from tweets: coffee_tweets

coffee_tweets <- tweets$text

head(coffee_tweets)
```

Make the vector a VCorpus object (1)

Recall that you've loaded your text data as a vector called `coffee_tweets` in the last exercise. Your next step is to convert this vector containing the text data to a corpus. As you've learned in the video, a corpus is a collection of documents, but it's also important to know that in the `tm` domain, R recognizes it as a data type.
There are two kinds of the corpus data type, the *permanent corpus*, `PCorpus`, and the *volatile corpus*, `VCorpus`. In essence, the difference between the two has to do with how the collection of documents is stored in your computer. In this course, we will use the volatile corpus, which is held in your 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, `coffee_tweets`, 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. Give it a shot!

```
# Load tm
library(tm)

# Make a vector source: coffee_source
coffee_source <- VectorSource(coffee_tweets)
```

Make the vector a VCorpus object (2)

Now that we've converted our vector to a Source object, we pass it to another `tm` function, `VCorpus()`, to create our volatile corpus. Pretty straightforward, right?
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`). It can help to **visualize** a `VCorpus` object to conceptualize the whole thing.
For example, to examine the contents of the second tweet in `coffee_corpus`, you'd subset twice. Once to specify the second `PlainTextDocument` corresponding to the

second tweet and again to extract the first (or `content`) element of that`PlainTextDocument`:

```
coffee_corpus[[15]][1]
```

coffee_source is already in your workspace

Make a volatile corpus: coffee_corpus

coffee_corpus <- VCorpus(coffee_source)

Print out coffee_corpus

coffee_corpus

Print data on the 15th tweet in coffee_corpus

coffee_corpus[[15]]

Print the content of the 15th tweet in coffee_corpus

coffee_corpus[[15]][1]

Make a VCorpus from a data frame

Print example_text to the console

example_text

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

# 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
```

Common cleaning functions from tm

Now that you know two ways to make a corpus, we can focus on cleaning, or preprocessing, the text. First, we'll clean a small piece of text so you can see how it works. Then we will move on to actual corpora.

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. Going forward, we'll load the `tm` and `qdap` for you when they are needed. Every time we introduce a new package, we'll have you load it the first time.

```
# 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)

# Remove punctuation
removePunctuation(text)

# Remove numbers
removeNumbers(text)

# Remove whitespace
stripWhitespace(text)
```

Cleaning with qdap

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")

```
## text is still loaded in your workspace

# Remove text within brackets
bracketX(text)

# Replace numbers with words
replace_number(text)

# Replace abbreviations
replace_abbreviation(text)

# Replace contractions
replace_contraction(text)

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

All about stop words

```
## text is preloaded into your workspace

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

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

# 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)
```

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`.
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.

```
# 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
```

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. Take a moment and try it out in the console. Be sure to include the punctuation marks.
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 punctation marks with the`removePunctuation()` function you learned a few exercises back. We then `strsplit()` this character vector of length 1 to length n,`unlist()`, then proceed to stem and re-complete.
Don't worry if that was confusing. Let's go through the process step by step!

```
# 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


# Re-complete stemmed document: complete_doc

complete_doc <- stemCompletion(stem_doc, comp_dict)
```

```
# Print complete_doc
complete_doc
```

Apply 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 like the one displayed in the editor, 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!*

```
# Alter the function code to match the instructions
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, c(stopwords("en"), "mug"))

  return(corpus)
}
```

```
# Apply your customized function to the tweet_corp: clean_corp
clean_corp  <- clean_corpus(tweet_corp)

# Print out a cleaned up tweet
clean_corp[[227]][1]

# Print out the same tweet in original form
tweets$text[[227]][1]
```

Understanding TDM and DTM

Make a document-term matrix

Hopefully you are not too tired after all this basic text mining work! Just in case, let's revisit the coffee tweets to build a document-term matrix.

Beginning with the `coffee.csv` file, we have used common transformations to produce a clean corpus called `clean_corp`.
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.

```
# Create the dtm from the corpus: coffee_dtm
coffee_dtm <- DocumentTermMatrix(clean_corp)

# Print out coffee_dtm data
coffee_dtm

# Convert coffee_dtm to a matrix: coffee_m
coffee_m <- as.matrix(coffee_dtm)
```

Make a term-document matrix

You're almost done with the not-so-exciting foundational work before we get to some fun visualizations and analyses based on the concepts you've learned so far!

In this exercise, you are performing a similar process but taking the*transpose* of the document-term matrix. In this case, the term-document matrix has terms in the first column and documents across the top as individual column names.

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.

```
# Create a TDM from clean_corp: coffee_tdm

coffee_tdm <- TermDocumentMatrix(clean_corp)


# Print coffee_tdm data

coffee_tdm


# Convert coffee_tdm to a matrix: coffee_m

coffee_m <- as.matrix(coffee_tdm)


# Print the dimensions of the matrix

dim(coffee_m)


# Review a portion of the matrix

coffee_m[2587:2590, 148:150]

# Print the dimensions of coffee_m

dim(coffee_m)


# Review a portion of the matrix
```

```
coffee_m[148:150, 2587:2590]
```

2 Word clouds and more interesting visuals

This chapter will teach you how to visualize text data in a way that's both informative and engaging.

Test your understanding of text mining

Words clouds help decision makers come to quick conclusions.

Frequent terms with tm

Now that you know how to make a term-document matrix, as well as its transpose, the document-term matrix, we will use it as the basis for some analysis. In order to analyze it we need to change it to a simple matrix like we did in chapter 1 using `as.matrix`. Calling `rowSums()` on your newly made matrix aggregates all the terms used in a passage. Once you have the `rowSums()`, you can`sort()` them with `decreasing = TRUE`, so you can focus on the most common terms.
Lastly, you can make a `barplot()` of the top 5 terms of`term_frequency` with the following code.

```
barplot(term_frequency[1:5], col = "#C0DE25")
```

Of course, you could take our ggplot2 course to learn how to customize the plot even more... :)

```
## coffee_tdm is still loaded in your workspace


# Create a matrix: coffee_m

coffee_m <- as.matrix(coffee_tdm)


# Calculate the rowSums: term_frequency

term_frequency <- rowSums(coffee_m)


# Sort term_frequency in descending order

term_frequency <- sort(term_frequency,

decreasing = TRUE)
```

```
# View the top 10 most common words
head(term_frequency,n=10)

# Plot a barchart of the 10 most common words
barplot(term_frequency[1:10],
col = "tan", las = 2)
```

Frequent terms with qdap

If you are OK giving up some control over the exact preproccessing steps, then a fast way to get frequent terms is with `freq_terms()` from `qdap`.
The function accepts a text variable, which in our case is the `tweets$text` vector. You can specify the top number of terms to show with the `top` argument, a vector of stop words to remove with the `stopwords` argument, and the minimum character length of a word to be included with the `at.least` argument. `qdap` has its own list of stop words that differ from those in `tm`. Our exercise will show you how to use either and compare their results.
Making a basic plot of the results is easy. Just call `plot()` on the `freq_terms()` object.

```
# Create frequency
frequency <- freq_terms(
tweets$text,
top = 10,
at.least = 3,
stopwords = "Top200Words"
)

# Make a frequency barchart
plot(frequency)

# Create frequency2
```

```
frequency2 <-  freq_terms(

tweets$text,

top = 10,

at.least = 3,

stopwords = tm::stopwords("english")

)


# Make a frequency2 barchart

plot(frequency2)
```

A simple word cloud

At this point you have had too much coffee. Plus, seeing the top words such as "shop", "morning", and "drinking" among others just isn't all that insightful.

In celebration of making it this far, let's try our hand on another batch of 1000 tweets. For now, you won't know what they have in common, but let's see if you can figure it out using a word cloud. The tweets' term-document matrix, matrix, and frequency values are preloaded in your workspace.

A word cloud is a visualization of terms. In a word cloud, size is often scaled to frequency and in some cases the colors may indicate another measurement. For now, we're keeping it simple: size is related to individual word frequency and we are just selecting a single color.

As you saw in the video, the `wordcloud()` function works like this:

```
wordcloud(words, frequencies, max.words = 500, colors = "blue")
```

Text mining analyses often include simple word clouds. In fact, they are probably over used, but can still be useful for quickly understanding a body of text!

```
## term_frequency is loaded into your workspace


# Load wordcloud package

library(wordcloud)


# Print the first 10 entries in term_frequency
```

```
head(term_frequency,n=10)

# Create word_freqs
word_freqs <- data.frame(term = names(term_frequency),
num = term_frequency)

# Create a wordcloud for the values in word_freqs
wordcloud(word_freqs$term, word_freqs$num,
max.words = 100, colors = "red")
```

Stop words and word clouds

```
# Add new stop words to clean_corpus()
clean_corpus <- function(corpus){
  corpus <- tm_map(corpus, removePunctuation)
  corpus <- tm_map(corpus, stripWhitespace)
  corpus <- tm_map(corpus, removeNumbers)
  corpus <- tm_map(corpus, content_transformer(tolower))
  corpus <- tm_map(corpus, removeWords,
                   c(stopwords("en"),"amp", "chardonnay", "wine", "glass" ))
  return(corpus)
}

# Create clean_chardonnay
clean_chardonnay <- clean_corpus(chardonnay_corp)

# Create chardonnay_tdm
chardonnay_tdm <- TermDocumentMatrix(clean_chardonnay)

# Create chardonnay_m
```

```
chardonnay_m <- as.matrix(chardonnay_tdm)


# Create chardonnay_words
chardonnay_words <- rowSums(chardonnay_m)
```

Plot the better word cloud

Now that you've added new stop words to the `clean_corpus()` function, let's take a look at the improved word cloud!
Your results from the previous exercise are preloaded into your workspace. Let's take a look at these new results.

```
# Sort the chardonnay_words in descending order
chardonnay_words <- sort(chardonnay_words,
decreasing = TRUE)


# Print the 6 most frequent chardonnay terms
head(chardonnay_words)


# Create chardonnay_freqs
chardonnay_freqs <-  data.frame(term = names(chardonnay_words),
num = chardonnay_words)


# Create a wordcloud for the values in word_freqs
wordcloud(chardonnay_freqs$term, chardonnay_freqs$num,
max.words = 50, colors = "red")
```

Improve word cloud colors

So far, you have specified only a single hexidecimal color to make your word clouds. You can easily improve the appearance of a word cloud. Instead of the `#AD1DA5` in the code below, you can specify a vector of colors to make certain words stand out or to fit an existing color scheme.

```
wordcloud(chardonnay_freqs$term,
          chardonnay_freqs$num,
          max.words = 100,
          colors = "#AD1DA5")
```

To change the `colors` argument of the`wordcloud()` function, you can use a vector of named colors like `c("chartreuse", "cornflowerblue", "darkorange")`. The function `colors()` will list all 657 named colors. You can also use this PDF as a reference.

```
# Print the list of colors

colors()



# Print the wordcloud with the specified colors

wordcloud(chardonnay_freqs$term,

      chardonnay_freqs$num,

      max.words = 100,

      colors = c("grey80", "darkgoldenrod1",  "tomato"))
```

Use prebuilt color palettes

In celebration of your text mining skills, you may have had too many glasses of chardonnay while listening to Marvin Gaye. If that's the case and you find yourself unable to pick good looking colors on your own, you can use the`RColorBrewer` package to help. `RColorBrewer`color schemes are organized into three categories:

- **Sequential**: Colors ascend from light to dark in sequence
- **Qualitative**: Colors are chosen for their pleasing qualities together
- **Diverging**: Colors have two distinct color spectra with lighter colors in between

To change the `colors` parameter of the `wordcloud()` function you can use a select a palette from `RColorBrewer` such as `"Greens"`. The function `display.brewer.all()` will list all predefined color palettes. More information on ColorBrewer (the framework behind `RColorBrewer`) is available on its website.

The function `brewer.pal()` allows you to select colors from a palette. Specify the number of distinct colors needed (e.g. 8) and the predefined palette to select from (e.g. `"Greens"`). Often in word clouds, very faint colors are washed out so it may make sense to remove the first couple from a `brewer.pal()` selection, leaving only the darkest.

Here's an example:

```
green_pal <- brewer.pal(8, "Greens")
green_pal <- green_pal[-(1:2)]
```

Then just add that object to the `wordcloud()` function.

```
wordcloud(chardonnay_freqs$term, chardonnay_freqs$num, max.words = 100,
colors = green_pal)
```

```
# List the available colors

display.brewer.all()



# Create purple_orange

purple_orange <- brewer.pal(10, "PuOr")



# Drop 2 faintest colors

purple_orange <- purple_orange[-(1:2)]



# Create a wordcloud with purple_orange palette

wordcloud(chardonnay_freqs$term, chardonnay_freqs$num, max.words = 100, colors = purple_orange)
```

Other word clouds and word networks
Find common words

Say you want to visualize common words across multiple documents. You can do this with `commonality.cloud()`.
Each of our coffee and chardonnay corpora is composed of many individual tweets. To treat the coffee tweets as a single document and likewise for chardonnay, you `paste()` together all the tweets in each corpus along with the parameter `collapse = " "`. This collapses all tweets (separated by a space) into a single vector. Then you can create a vector containing the two collapsed documents.

```
all_coffee <- paste(coffee$tweets, collapse = " ")
all_chardonnay <- paste(chardonnay$tweets, collapse = " ")
all_tweets <- c(all_coffee, all_chardonnay)
```

Once you're done with these steps, you can take the same approach you've seen before to create a `VCorpus()` based on a`VectorSource` from the `all_tweets` object.

```
# Create all_coffee

all_coffee <- paste(coffee_tweets$text, collapse = " ")



# Create all_chardonnay

all_chardonnay <- paste(chardonnay_tweets$text, collapse = " ")



# Create all_tweets

all_tweets <- c(all_coffee, all_chardonnay)



# Convert to a vector source

all_tweets <- VectorSource(all_tweets)



# Create all_corpus

all_corpus <- VCorpus(all_tweets)
```

Visualize common words

Now that you have a corpus filled with words used in both the chardonnay and coffee tweets files, you can clean the corpus, convert it into a `TermDocumentMatrix`, and then a matrix to prepare it for a `commonality.cloud()`
The `commonality.cloud()` function accepts this matrix object, plus additional arguments like `max.words` and `colors` to further customize the plot.

```
commonality.cloud(tdm_matrix, max.words = 100, colors = "springgreen")
```

```
# Clean the corpus

all_clean <- clean_corpus(all_corpus)



# Create all_tdm

all_tdm <- TermDocumentMatrix(all_clean)



# Create all_m

all_m <- as.matrix(all_tdm)



# Print a commonality cloud

commonality.cloud(all_m, max.words = 100, colors = "steelblue1")
```

Visualize dissimilar words

Say you want to visualize the words not in common. To do this, you can also use `comparison.cloud()` and the steps are quite similar with one main difference.
Like when you were searching for words in common, you start by unifying the tweets into distinct corpora and combining them into their own `VCorpus()` object. Next apply a `clean_corpus()` function and organize it into a `TermDocumentMatrix`.
To keep track of what words belong to `coffee` versus `chardonnay`, you can set the column names of the TDM like this:

```
colnames(all_tdm) <- c("chardonnay", "coffee")
```

Lastly, convert the object to a matrix using `as.matrix()` for use in `comparison.cloud()`. For every distinct corpora passed to the `comparison.cloud()` you can specify a color as in `colors = c("red", "yellow", "green")` to make the sections distinguishable.

```
# Clean the corpus

all_clean <- clean_corpus(all_corpus)



# Create all_tdm

all_tdm <- TermDocumentMatrix(all_clean)



# Give the columns distinct names

colnames(all_tdm) <- c("coffee", "chardonnay")



# Create all_m

all_m <- as.matrix(all_tdm)



# Create comparison cloud

comparison.cloud(all_m, max.words = 50, colors = c("orange", "blue"))
```

Polarized tag cloud

A `commonality.cloud()` may be misleading since words could be represented disproportionately in one corpus or the other, even if they are shared. In the commonality cloud, they would show up without telling you which one of the corpora has more term occurrences. To solve this problem, we can create a `pyramid.plot()` from the `plotrix` package.

Building on what you already know, we have created a simple matrix from the coffee and chardonnay tweets using `all_tdm_m <- as.matrix(all_tdm)`. Recall that this matrix contains two columns: one for term frequency in the chardonnay corpus, and another for term frequency in the coffee corpus. So we can use the `subset()` function in the following way to get terms that appear one or more times in both corpora:

```
same_words <- subset(all_tdm_m, all_tdm_m[, 1] > 0 & all_tdm_m[, 2] > 0)
```

Once you have the terms that are common to both corpora, you can create a new column in `same_words` that contains the absolute difference between how often each term is used in each corpus.

To identify the words that differ the most between documents, we must `order()` the rows of `same_words` by the absolute difference column with `decreasing = TRUE` like this:

```
same_words <- same_words[order(same_words[, 3], decreasing = TRUE), ]
```

Now that `same_words` is ordered by the absolute difference, let's create a small `data.frame()` of the 20 top terms so we can pass that along to `pyramid.plot()`:

```
top_words <- data.frame(
  x = same_words[1:20, 1],
  y = same_words[1:20, 2],
  labels = rownames(same_words[1:20, ])
)
```

Note that `top_words` contains columns `x` and `y` for the frequency of the top words for each of the documents, and a third column, `labels`, that contains the words themselves. Finally, you can create your `pyramid.plot()` and get a better feel for how the word usages differ by topic!

```
pyramid.plot(top_words$x, top_words$y,
             labels = top_words$labels, gap = 8,
             top.labels = c("Chardonnay", "Words", "Coffee"),
             main = "Words in Common", laxlab = NULL,
             raxlab = NULL, unit = NULL)
```

```
# Create common_words

common_words <- subset(all_tdm_m, all_tdm_m[, 1] > 0 & all_tdm_m[, 2] > 0)



# Create difference

difference <- abs(common_words[, 1] - common_words[, 2])



# Combine common_words and difference

common_words <- cbind(common_words, difference)



# Order the data frame from most differences to least

common_words <- common_words[order(common_words[, 3], decreasing = TRUE), ]



# Create top25_df
```

```
top25_df <- data.frame(x = common_words[1:25, 1],
                       y = common_words[1:25, 2],
                       labels = rownames(common_words[1:25, ]))

# Create the pyramid plot
pyramid.plot(top25_df$x, top25_df$y, labels = top25_df$labels,
             gap = 8, top.labels = c("Chardonnay", "Words", "Coffee"),
             main = "Words in Common", laxlab = NULL,
             raxlab = NULL, unit = NULL)
```

Visualize word networks

```
# Word association
word_associate(coffee_tweets$text, match.string = c("barista"),
               stopwords = c(Top200Words, "coffee", "amp"),
               network.plot = TRUE, cloud.colors = c("gray85", "darkred"))

# Add title
title(main = "Barista Coffee Tweet Associations")
```

Teaser: simple word clustering

```
# Plot a dendrogram
plot(hc)
```

3 Adding to your tm skills

In this chapter, you'll learn more basic text mining techniques based on the bag of words method.

Distance matrix and dendrogram

A simple way to do word cluster analysis is with a dendrogram on your term-document matrix. Once you have a TDM, you can call `dist()` to compute the differences between each row of the matrix.
Next, you call `hclust()` to perform cluster analysis on the dissimilarities of the distance matrix. Lastly, you can visualize the word frequency distances using a dendrogram and `plot()`. Often in text mining, you can tease out some interesting insights or word clusters based on a dendrogram.
Consider the table of annual rainfall that you saw in the last video. Cleveland and Portland have the same amount of rainfall, so their distance is 0. You might expect the two cities to be a cluster and for New Orleans to be on its own since it gets vastly more rain.

```
       city rainfall
  Cleveland    39.14
   Portland    39.14
     Boston    43.77
New Orleans    62.45
```

```
# Create dist_rain

dist_rain <- dist(rain[, 2])



# View the distance matrix

dist_rain



# Create hc

hc <- hclust(dist_rain)



# Plot hc

plot(hc, labels = rain$city)
```

Make a distance matrix and dendrogram from a TDM

Now that you understand the steps in making a dendrogram, you can apply them to text. But first, you have to limit the number of words in your TDM using `removeSparseTerms()` from `tm`. Why would you want to adjust the sparsity of the TDM/DTM?

TDMs and DTMs are sparse, meaning they contain mostly zeros. Remember that 1000 tweets can become a TDM with over 3000 terms! You won't be able to easily interpret a dendrogram that is so cluttered, especially if you are working on more text.

A good TDM has between 25 and 70 terms. The lower the `sparse` value, the more terms are kept. The closer it is to 1, the fewer are kept. This value is a percentage cutoff of zeros for each term in the TDM.

```
# Print the dimensions of tweets_tdm

dim(tweets_tdm)


# Create tdm1

tdm1 <- removeSparseTerms(tweets_tdm,sparse=0.95)


# Create tdm2

tdm2 <- removeSparseTerms(tweets_tdm,sparse=0.975)


# Print tdm1

tdm1


# Print tdm2

tdm2
```

Put it all together: a text based dendrogram

Its time to put your skills to work to make your first text-based dendrogram. Remember, dendrograms reduce information to help you make sense of the data. This is much like how an average tells you something, but not everything, about a population. Both can be misleading. With text, there are often a lot of nonsensical clusters, but some valuable clusters may also appear.

A peculiarity of TDM and DTM objects is that you have to convert them first to matrices (with `as.matrix()`), then to data frames (with`as.data.frame()`), before using them with the `dist()` function.
For the chardonnay tweets, you may have been surprised to see the soul music legend Marvin Gaye appear in the word cloud. Let's see if the dendrogram picks up the same.

```
# Create tweets_tdm2

tweets_tdm2 <- removeSparseTerms(tweets_tdm,sparse = 0.975)


# Create tdm_m

tdm_m <- as.matrix(tweets_tdm2)


# Create tdm_df

tdm_df <- as.data.frame(tdm_m)


# Create tweets_dist

tweets_dist <- dist(tdm_df)


# Create hc

hc <- hclust(tweets_dist)


# Plot the dendrogram
```

```
plot(hc)
```

Dendrogram aesthetics

So you made a dendrogram...but its not as eye catching as you had hoped!

The `dendextend` package can help your audience by coloring branches and outlining clusters. `dendextend` is designed to operate on dendrogram objects, so you'll have to change the hierarchical cluster from `hclust` using `as.dendrogram()`.
A good way to review the terms in your dendrogram is with the`labels()` function. It will print all terms of the dendrogram. To highlight specific branches, use `branches_attr_by_labels()`. First, pass in the dendrogram object, then a vector of terms as in`c("data", "camp")`. Lastly add a color such as `"blue"`.
After you make your plot, you can call out clusters with`rect.dendrogram()`. This adds rectangles for each cluster. The first argument to `rect.dendrogram()` is the dendrogram, followed by the number of clusters (`k`). You can also pass a `border` argument specifying what color you want the rectangles to be (e.g. `"green"`).

```
# Load dendextend

library(dendextend)



# Create hc

hc <- hclust(tweets_dist)



# Create hcd

hcd <- as.dendrogram(hc)



# Print the labels in hcd

labels(hcd)



# Change the branch color to red for "marvin" and "gaye"

hcd <- branches_attr_by_labels(hcd,
```

```
c("marvin", "gaye"), "red")


# Plot hcd

plot(hcd, main = "Better Dendrogram")


# Add cluster rectangles

rect.dendrogram(hcd, k = 2, border = "grey50")
```

Using word association

Another way to think about word relationships is with the `findAssocs()` function in the `tm` package. For any given word, `findAssocs()` calculates its correlation with every other word in a TDM or DTM. Scores range from 0 to 1. A score of 1 means that two words always appear together, while a score of 0 means that they never appear together.

To use `findAssocs()` pass in a TDM or DTM, the search term, and a minimum correlation. The function will return a list of all other terms that meet or exceed the minimum threshold.

```
findAssocs(tdm, "word", 0.25)
```

Minimum correlation values are often relatively low because of word diversity. Don't be surprised if `0.10` demonstrates a strong pairwise term association.

The coffee tweets have been cleaned and organized into `tweets_tdm` for the exercise. You will search for a term association, and manipulate the results with `list_vect2df()` from `qdap` and then create a plot with the `ggplot2` code in the example script.

```
# Create associations

associations <- findAssocs(tweets_tdm , "venti", 0.2)


# View the venti associations

associations
```

```
# Create associations_df

associations_df <- list_vect2df(associations)[, 2:3]


# Plot the associations_df values (don't change this)

ggplot(associations_df, aes(y = associations_df[, 1])) +

  geom_point(aes(x = associations_df[, 2]),

             data = associations_df, size = 3) +

  theme_gdocs()
```

N-gram tokenization

Will increasing the n-gram length increase, decrease or make no difference for the TDM or DTM size?

Increase

Changing n-grams

So far, we have only made TDMs and DTMs using single words. The default is to make them with unigrams, but you can also focus on tokens containing two or more words. This can help extract useful phrases which lead to some additional insights or provide improved predictive attributes for a machine learning algorithm.

The function below uses the `RWeka` package to create trigram (three word) tokens: `min` and `max` are both set to `3`.

```
tokenizer <- function(x)
  NGramTokenizer(x, Weka_control(min = 3, max = 3))
```

Then the customized `tokenizer()` function can be passed into the`TermDocumentMatrix` or `DocumentTermMatrix` functions as an additional parameter:

```
tdm <- TermDocumentMatrix(
  corpus,
  control = list(tokenize = tokenizer)
)
```

```
# Make tokenizer function

tokenizer <- function(x)
```

```
  NGramTokenizer(x, Weka_control(min = 2, max = 2))


# Create unigram_dtm

unigram_dtm <- DocumentTermMatrix(text_corp)


# Create bigram_dtm

bigram_dtm <- DocumentTermMatrix(

  text_corp,

  control = list(tokenize = tokenizer)

)


# Examine unigram_dtm

unigram_dtm


# Examine bigram_dtm

bigram_dtm
```

How do bigrams affect word clouds?

Now that you have made a bigram DTM, you can examine it and remake a word cloud. The new tokenization method affects not only the matrices, but also any visuals or modeling based on the matrices.

Remember how "Marvin" and "Gaye" were separate and large terms in the chardonnay word cloud? Using bigram tokenization grabs all two word combinations. Observe what happens to the word cloud in this exercise.

```
# Create bigram_dtm_m

bigram_dtm_m <- as.matrix(bigram_dtm)
```

```
# Create freq

freq <- colSums(bigram_dtm_m)


# Create bi_words

bi_words <- names(freq)


# Examine part of bi_words

bi_words[2577:2587]


# Plot a wordcloud

wordcloud(bi_words, freq,

max.words = 15, colors = "red")
```

Different frequency criteria
Changing frequency weights

So far you have used term frequency to make the`DocumentTermMatrix` or `TermDocumentMatrix`. There are other term weights that can be helpful. The most popular weight is `TfIdf`, which stands for *term frequency-inverse document frequency*.
The `TfIdf` score increases by term occurrence but is penalized by the frequency of appearance among all documents.
From a common sense perspective, if a term appears often it must be important. This attribute is represented by term frequency (i.e. `Tf`), which is normalized by the length of the document. However, if the term appears in all documents, it is not likely to be insightful. This is captured in the inverse document frequency (i.e. `Idf`).
The wiki page on `TfIdf` contains the mathematical explanation behind the score, but the exercise will demonstrate the practical difference.

```
# Create tf_tdm

tf_tdm <- TermDocumentMatrix(text_corp)
```

```
# Create tfidf_tdm

tfidf_tdm <- TermDocumentMatrix(text_corp,control = list(weighting = weightTfIdf))


# Create tf_tdm_m

tf_tdm_m <- as.matrix(tf_tdm)


# Create tfidf_tdm_m

tfidf_tdm_m <- as.matrix(tfidf_tdm)


# Examine part of tf_tdm_m

tf_tdm_m[508:509, 5:10]


# Examine part of tfidf_tdm_m

tfidf_tdm_m[508:509, 5:10]
```

Capturing metadata in tm

Depending on what you are trying to accomplish, you may want to keep metadata about the document when you create a TDM or DTM. This metadata can be incorporated into the corpus fairly easily by creating a `readerControl` list and applying it to a `DataframeSource` when calling `VCorpus()`.

You will need to know the column names of the data frame containing the metadata to be captured. The `names()` function is helpful for this.

To capture the text column of the coffee tweets `text` along with a metadata column of unique numbers called `num` you would use the code below.

```
custom_reader <- readTabular(
  mapping = list(content = "text", id = "num")
)
text_corpus <- VCorpus(
  DataframeSource(tweets),
```

```
  readerControl = list(reader = custom_reader)
)
```

```
# Add author to custom reading list

custom_reader <- readTabular(

  mapping = list(content = "text",

                 id = "num",

                 author = "screenName",

                 date = "created")

)


# Make corpus with custom reading

text_corpus <- VCorpus(

  DataframeSource(tweets),

  readerControl = list(reader = custom_reader)

)


# Clean corpus

text_corpus <- clean_corpus(text_corpus)


# Print data

text_corpus[[1]][1]


# Print metadata

text_corpus[[1]][2]
```

4 Battle of the tech giants for talent

Amazon vs. Google

Step 1: Problem definition

Does Amazon or Google have a better perceived pay according to online reviews?

Does Amazon or Google have a better work-life balance according to current employees?

Step 2: Identifying the text sources

Employee reviews can come from various sources. If your human resources department had the resources, you could have a third party administer focus groups to interview employees both internally and from your competitor.

Forbes and others publish articles about the "best places to work", which may mention Amazon and Google. Another source of information might be anonymous online reviews from websites like Indeed, Glassdoor orCareerBliss.

Here, we'll focus on a collection of anonymous online reviews.

```
# Print the structure of amzn

str(amzn)


# Create amzn_pros

amzn_pros <- amzn$pros


# Create amzn_cons

amzn_cons <- amzn$cons


# Print the structure of goog

str(goog)


# Create goog_pros
```

```
goog_pros <- goog$pros


# Create goog_cons

goog_cons <- goog$cons
```

Text organization

Now that you have selected the exact text sources, you are ready to clean them up. You'll be using the two functions you just saw in the video: `qdap_clean()`, which applies a series of `qdap` functions to a text vector, and `tm_clean()`, which applies a series of `tm` functions to a corpus object. You can refer back to the video to remind yourself of how they work.

In order to keep things simple, the functions have been defined for you and are available in your workspace. It's your job to apply them to `amzn_pros` and `amzn_cons`!

```
# qdap cleaning function

qdap_clean <- function(x) {

x <- replace_abbreviation(x)

x <- replace_contraction(x)

x <- replace_number(x)

x <- replace_ordinal(x)

x <- replace_symbol(x)

x <- tolower(x)

return(x)

}


# tm cleaning function
tm_clean <- function(corpus) {
tm_clean <- tm_map(corpus, removePunctuation)
corpus <- tm_map(corpus, stripWhitespace)
corpus <- tm_map(corpus, removeWords,
```

```
c(stopwords("en"), "Google", "Amazon", "company"))
return(corpus)
}


# Alter amzn_pros

amzn_pros <- qdap_clean(amzn_pros)


# Alter amzn_cons

amzn_cons <- qdap_clean(amzn_cons)


# Create az_p_corp

az_p_corp <- VCorpus(VectorSource(amzn_pros))


# Create az_c_corp

az_c_corp <- VCorpus(VectorSource(amzn_cons))


# Create amzn_pros_corp

amzn_pros_corp <- tm_clean(az_p_corp)


# Create amzn_cons_corp

amzn_cons_corp <- tm_clean(az_c_corp)
```

Working with Google reviews

```
# Apply qdap_clean to goog_pros

goog_pros <- qdap_clean(goog_pros)
```

```
# Apply qdap_clean to goog_cons

goog_cons <- qdap_clean(goog_cons)


# Create goog_p_corp

goog_p_corp <- VCorpus(VectorSource(goog_pros))


# Create goog_c_corp

goog_c_corp <- VCorpus(VectorSource(goog_cons))


# Create goog_pros_corp

goog_pros_corp <- tm_clean(goog_p_corp)


# Create goog_cons_corp

goog_cons_corp <- tm_clean(goog_c_corp)
```

Steps 4 & 5: Feature extraction & analysis

Feature extraction & analysis: amzn_pros

`amzn_pros_corp`, `amzn_cons_corp`, `goog_pros_corp` and `goog_cons_corp` have all been preprocessed, so now you can extract the features you want to examine. Since you are using the bag of words approach, you decide to create a bigram `TermDocumentMatrix` for Amazon's positive reviews corpus, `amzn_pros_corp`. From this, you can quickly create a `wordcloud()` to understand what phrases people positively associate with working at Amazon.

The function below uses `RWeka` to tokenize two terms and is used behind the scenes in this exercise.

```
tokenizer <- function(x)
  NGramTokenizer(x, Weka_control(min = 2, max = 2))
```

```
# Create amzn_p_tdm

amzn_p_tdm <- TermDocumentMatrix(amzn_pros_corp,control = list(tokenize = tokenizer))


# Create amzn_p_tdm_m

amzn_p_tdm_m <- as.matrix(amzn_p_tdm)


# Create amzn_p_freq

amzn_p_freq <- rowSums(amzn_p_tdm_m)


# Plot a wordcloud using amzn_p_freq values

wordcloud(names(amzn_p_freq), amzn_p_freq ,

max.words = 25, colors = "blue")
```

Feature extraction & analysis: amzn_cons

```
# Create amzn_c_tdm

amzn_c_tdm <- TermDocumentMatrix(amzn_cons_corp ,control = list(tokenize = tokenizer))


# Create amzn_c_tdm_m

amzn_c_tdm_m <- as.matrix(amzn_c_tdm)


# Create amzn_c_freq

amzn_c_freq <- rowSums(amzn_c_tdm_m)


# Plot a wordcloud of negative Amazon bigrams

wordcloud(names(amzn_c_freq), amzn_c_freq ,

max.words = 25, colors = "red")
```

amzn_cons dendrogram

It seems there is a strong indication of long working hours and poor work-life balance in the reviews. As a simple clustering technique, you decide to perform a hierarchical cluster and create a dendrogram to see how connected these phrases are.

```
# Create amzn_c_tdm

amzn_c_tdm <- TermDocumentMatrix(amzn_cons_corp,control = list(tokenize = tokenizer))


# Print amzn_c_tdm to the console

amzn_c_tdm


# Create amzn_c_tdm2 by removing sparse terms

amzn_c_tdm2 <- removeSparseTerms(amzn_c_tdm ,sparse=0.993)


# Create hc as a cluster of distance values

hc <- hclust(dist(amzn_c_tdm2, method = "euclidean"),method = "complete")


# Produce a plot of hc

plot(hc)
```

Word association

As expected, you see similar topics throughout the dendrogram. Switching back to positive comments, you decide to examine top phrases that appeared in the word clouds. You hope to

find associated terms using the `findAssocs()` function from `tm`. You want to check for something surprising now that you have learned of long hours and a lack of work-life balance.

```
# Create amzn_p_tdm

amzn_p_tdm <- TermDocumentMatrix(amzn_pros_corp ,control = list(tokenize = tokenizer))


# Create amzn_p_m

amzn_p_m <- as.matrix(amzn_p_tdm)


# Create amzn_p_freq

amzn_p_freq <- rowSums(amzn_p_m)


# Create term_frequency

term_frequency <- sort(amzn_p_freq,decreasing = TRUE)


# Print the 5 most common terms

head(term_frequency,n=5)


# Find associations with fast paced

findAssocs(amzn_p_tdm , "fast paced",0.2)
```

Quick review of Google reviews

You decide to create a `comparison.cloud()` of Google's positive and negative reviews for comparison to Amazon. This will give you a quick understanding of top terms without having to spend as much time as you did examining the Amazon reviews in the previous exercises.

We've provided you with a corpus `all_goog_corpus`, which has the 500 positive and 500 negative reviews for Google. Here, you'll clean the corpus and create a comparison cloud comparing the common words in both pro and con reviews.

```
# Create all_goog_corp

all_goog_corp <- tm_clean(all_goog_corpus)


# Create all_tdm

all_tdm <- TermDocumentMatrix(all_goog_corp)


# Name the columns of all_tdm

colnames(all_tdm) <- c("Goog_Pros", "Goog_Cons")


# Create all_m

all_m <- as.matrix(all_tdm)


# Build a comparison cloud

comparison.cloud(all_m,colors = c("#F44336", "#2196f3"),max.words = 100)
```

Cage match! Amazon vs. Google pro reviews

Amazon's positive reviews appear to mention bigrams such as "good benefits", while its negative reviews focus on bigrams such as "work load" and "work-life balance" issues.

In contrast, Google's positive reviews mention "great food", "perks", "smart people", and "fun culture", among other things. Google's negative reviews discuss "politics", "getting big", "bureaucracy", and "middle management".

You decide to make a pyramid plot lining up positive reviews for Amazon and Google so you can adequately see the differences between any shared birgrams.

```
# Create common_words
```

```
common_words <- subset(all_tdm_m, all_tdm_m[, 1] > 0 & all_tdm_m[, 2] > 0)

# Create difference
difference <- abs(common_words[, 1] - common_words[, 2])

# Add difference to common_words
common_words <- cbind(common_words, difference)

# Order the data frame from most differences to least
common_words <- common_words[order(common_words[, 3], decreasing = TRUE), ]

# Create top15_df
top15_df <- data.frame(x = common_words[1:15, 1],
                       y = common_words[1:15, 2],
                       labels = rownames(common_words[1:15, ]))

# Create the pyramid plot
pyramid.plot(top15_df$x, top15_df$y,
             labels = top15_df$labels, gap = 12,
             top.labels = c("Amzn", "Pro Words", "Google"),
             main = "Words in Common", unit = NULL)
```

Cage match, part 2! Negative reviews

Interestingly, some Amazon employees discussed "work-life balance" as a positive. In both organizations, people mentioned "culture" and "smart people", so there are some similar positive aspects between the two companies.

You now decide to turn your attention to negative reviews and make the same visual. This time,`all_tdm_m` contains the negative reviews, or cons, from both organizations.

```
# Create common_words

common_words <- subset(all_tdm_m, all_tdm_m[, 1] > 0 & all_tdm_m[, 2] > 0)


# Create difference

difference <- abs(common_words[, 1] - common_words[, 2])


# Bind difference to common_words

common_words <- cbind(common_words, difference)


# Order the data frame from most differences to least

common_words <- common_words[order(common_words[, 3], decreasing = TRUE), ]


# Create top15_df

top15_df <- data.frame(x = common_words[1:15, 1],

                       y = common_words[1:15, 2],

                       labels = rownames(common_words[1:15, ]))


# Create the pyramid plot

pyramid.plot(top15_df$x, top15_df$y,

          labels = top15_df$labels, gap = 12,
```

```
top.labels = c("Amzn", "Cons Words", "Google"),
main = "Words in Common", unit = NULL)
```

Draw conclusions, insights, or recommendations

Based on the visual, does Amazon or Google have a better work-life balance according to current employee reviews?

Google

Draw another conclusion, insight, or recommendation

Earlier you were surprised to see "fast paced" in the pros despite the other reviews mentioning "work-life balance". Recall that you used `findAssocs()` to get a named vector of phrases. These may lead you to a conclusion about the type of person who favorably views an intense workload.
Given the abbreviated results of the associated phrases, what would you recommend Amazon HR recruiters look for in candidates? (You can use the snippet below to gain insight on phrases associated with "fast paced".)

```
findAssocs(amzn_p_tdm, "fast paced", 0.2)[[1]][1:15]
```

Identify candidates that view an intense workload as an opportunity to learn fast and give them ample opportunity.