Introduction to Text Mining Using R

Ritu Sanjay

**Quick taste of text mining**

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 the 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)

**Instructions**

**100 XP**

We've created an object in your workspace called new\_text containing several sentences.

* Load the qdap package.
* Print new\_text to the console.
* Create term\_count consisting of the 10 most frequent terms in new\_text.
* Plot a bar chart with the results of term\_count.

[**Take Hint (-30 XP)**](javascript:void(0))

# Load qdap

library(qdap)

# Print new\_text to the console

print(new\_text)

# Find the 10 most frequent terms: term\_count

term\_count <- freq\_terms(new\_text,10)

# Plot term\_count

plot(term\_count)

**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.

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).

A word of warning: you'll be working with real tweets from real people in this course, so you may find some mild profanity from time to time.

**Instructions**

**100 XP**

The data has been loaded for you and is available in coffee\_data\_file.

* Create a new object tweets using read.csv() on the file coffee\_data\_file, which contains tweets mentioning coffee. Remember to add stringsAsFactors = FALSE!
* Examine the tweets object using str() to determine which column has the text you'll want to analyze.
* Make a new coffee\_tweets object using only the text column you identified earlier. To do so, use the $ operator and column name.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import text data from CSV, no factors

tweets <- read.csv(coffee\_data\_file, stringsAsFactors = FALSE)

# View the structure of tweets

str(tweets)

# Isolate text from tweets

coffee\_tweets <- tweets$text

**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!

**Instructions**

**100 XP**

* Load the tm package.
* Create a Source object from the coffee\_tweets vector. Call this new object coffee\_source.

[**Take Hint (-30 XP)**](javascript:void(0))

# Load tm

library(tm)

# Make a vector source from coffee\_tweets

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 a list containing actual text data (content), and some corresponding metadata (meta). It can help to [**visualize**](http://s3.amazonaws.com/assets.datacamp.com/production/course_935/datasets/vcorpus_visual.png) a VCorpus object to conceptualize the whole thing.

To review a single document object (the 10th) you subset with double square brackets.

coffee\_corpus[[10]]

To review the actual *text* you index the list twice. To access the document's metadata, like timestamp, change [1] to [2]. Another way to review the plain *text* is with the content() function which doesn't need the second set of brackets.

coffee\_corpus[[10]][1]

content(coffee\_corpus[[10]])

**Instructions**

**100 XP**

* Call the VCorpus() function on the coffee\_source object to create coffee\_corpus.
* Verify coffee\_corpus is a VCorpus object by printing it to the console.
* Print the 15th element of coffee\_corpus to the console to verify that it's a PlainTextDocument that contains the content and metadata of the 15th tweet. Use double bracket subsetting.
* Print the content of the 15th tweet in coffee\_corpus. Use double brackets to select the proper tweet, followed by single brackets to extract the content of that tweet.
* Print the content() of the 10th tweet within coffee\_corpus

[**Take Hint (-30 XP)**](javascript:void(0))

## coffee\_source is already in your workspace

# Make a volatile corpus from coffee\_corpus

coffee\_corpus <- VCorpus(coffee\_source)

# Print out coffee\_corpus

print(coffee\_corpus)

# Print the 15th tweet in coffee\_corpus

coffee\_corpus[[15]]

# Print the contents of the 15th tweet in coffee\_corpus

coffee\_corpus[[15]]

# Now use content to review plain text

content(coffee\_corpus[[10]])

**Make a VCorpus from a data frame**

If your text data is in a data frame you can use DataframeSource() for your analysis. The data frame passed to DataframeSource() must have a specific structure:

* Column **one** must be called doc\_id and contain a unique string for each row.
* Column **two** must be called text with "UTF-8" encoding (pretty standard).
* Any other columns, **3+** are considered metadata and will be retained as such.

This exercise introduces meta() to extract the metadata associated with each document. Often your data will have metadata such as authors, dates, topic tags or places which can inform your analysis. Once your text is a corpus, you can apply meta() to examine the additional document level information.

**Instructions**

**100 XP**

In your workspace, there's a simple data frame called example\_text with the correct column names and some metadata. There is also vec\_corpus which is a volatile corpus made with VectorSource()

* Create df\_source using DataframeSource() with the example\_text.
* Create df\_corpus by converting df\_source to a *volatile* corpus object with VCorpus().
* Print out df\_corpus. Notice how many documents it contains and the number of retained document level metadata points.
* Use meta() on df\_corpus to print the document associated metadata.
* Examine the pre-loaded vec\_corpus object. Compare the number of *documents* to df\_corpus.
* Use meta() on vec\_corpus to compare any metadata found between vec\_corpus and df\_corpus.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create a DataframeSource from the example text

df\_source <- DataframeSource(example\_text)

# Convert df\_source to a volatile corpus

df\_corpus <- VCorpus(df\_source)

# Examine df\_corpus

df\_corpus

# Examine df\_corpus metadata

meta(df\_corpus)

# Compare the number of documents in the vector source

vec\_corpus

# Compare metadata in the vector corpus

meta(vec\_corpus)

**Common cleaning functions from tm**

Now that you know two ways to make a corpus, you can focus on cleaning, or preprocessing, the text. First, you'll clean a small piece of text, then you will move on to larger corpora.

In bag of words text mining, cleaning helps aggregate terms. For example, it might make sense for the words "miner", "mining" and "mine" to 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

tolower() is part of base R, while the other three functions come from the tm package. Going forward, we'll load 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.

The variable text, containing a sentence, is shown in the script.

**Instructions**

**100 XP**

Apply each of the following functions to text, simply printing results to the console:

- `tolower()`

- `removePunctuation()`

- `removeNumbers()`

- `stripWhitespace()`

[**Take Hint (-30 XP)**](javascript:void(0))

# 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."

# Make lowercase

tolower(text)

# Remove punctuation

removePunctuation(text)

# Remove numbers

removePunctuation(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")

**Instructions**

**100 XP**

Apply the following functions to the text object from the previous exercise:

* bracketX()
* replace\_number()
* replace\_abbreviation()
* replace\_contraction()
* replace\_symbol()

[**Take Hint (-30 XP)**](javascript:void(0))

## 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

Often there are words that are frequent but provide little information. These are called stop words, and you may want to remove them from your analysis. Some common English stop words include "I", "she'll", "the", etc. In the tm package, there are 174 common English stop words (you'll print them in this exercise!)

When you are doing an analysis you will likely need to add to this list. In our coffee tweet example, all tweets contain "coffee", so it's important to pull out that word in addition to the common stop words. Leaving "coffee" in doesn't add any insight and will cause it to be overemphasized in a frequency analysis.

Using the c() function allows you to add new words 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.

##### Instructions

**100 XP**

* Review standard stop words by calling stopwords("en").
* Remove "en" stopwords from text.
* Add "coffee" and "bean" to the standard stop words, assigning to new\_stops.
* Remove the customized stopwords, new\_stops, from text.

[**Take Hint (-30 XP)**](javascript:void(0))

## 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. Word stemming reduces words to unify across documents. For example, the stem of "computational", "computers" and "computation" is "comput". But because "comput" isn't a real word, we want to re-construct the words so that "computational", "computers", and "computation" all refer a recognizable word, such as "computer". The reconstruction step is called 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".

You will use stemCompletion() to reconstruct these word roots back into a known term. stemCompletion() accepts a character vector and a 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" so all instances of "comput" can be reconstructed.

##### Instructions

**100 XP**

* Create a vector called complicate consisting of the words "complicated", "complication", and "complicatedly" in that order.
* Store the stemmed version of complicate to an object called stem\_doc.
* Create comp\_dict that contains one word, "complicate".
* Create complete\_text by applying stemCompletion() to stem\_doc. Re-complete the words using comp\_dict as the reference corpus.
* Print complete\_text to the console.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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 punctuation 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!

##### Instructions

**100 XP**

The document text\_data and the completion dictionary comp\_dict are loaded in your workspace.

* Remove the punctuation marks in text\_data using removePunctuation(), assigning to rm\_punc.
* Call strsplit() on rm\_punc with the split argument set equal to " ". Nest this inside unlist(), assigning to n\_char\_vec.
* Use stemDocument() again to perform word stemming on n\_char\_vec, assigning to stem\_doc.
* Create complete\_doc by re-completing your stemmed document with stemCompletion() and using comp\_dict as your reference corpus.

Are *stem\_doc* and *complete\_doc* what you expected?

[**Take Hint (-30 XP)**](javascript:void(0))

# 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 function tm\_map() to apply cleaning functions to an entire corpus, making the cleaning steps easier.

tm\_map() takes two arguments, a corpus and a cleaning function. Here, removeNumbers() is from the tm package.

corpus <- tm\_map(corpus, removeNumbers)

For compatibility, base R and qdap functions need to be wrapped in content\_transformer().

corpus <- tm\_map(corpus, content\_transformer(replace\_abbreviation))

You may be applying the same functions over multiple corpora; using a custom function like the one displayed in the editor will save you time (and lines of code). clean\_corpus() takes one argument, corpus, and applies a series of cleaning functions to it in order, then returns the updated corpus.

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 your results!

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Edit the custom function clean\_corpus() in the sample code to apply (in order):
  + tm's removePunctuation().
  + Base R's tolower().
  + Append "mug" to the stop words list.
  + tm's stripWhitespace().
* Create clean\_corp by applying clean\_corpus() to the included corpus tweet\_corp.
* Print the cleaned 227th tweet in clean\_corp using indexing [[227]] and content().
* Compare it to the original tweet from tweets$text tweet using [227].
  + [**Take Hint (-15 XP)**](javascript:void(0))

# Alter the function code to match the instructions

clean\_corpus <- function(corpus) {

# Remove punctuation

corpus <- tm\_map(corpus, removePunctuation)

# Transform to lower case

corpus <- tm\_map(corpus, content\_transformer(tolower))

# Add more stopwords

corpus <- tm\_map(corpus, removeWords, words = c(stopwords("en"), "coffee", "mug"))

# Strip whitespace

corpus <- tm\_map(corpus, stripWhitespace)

return(corpus)

}

# Apply your customized function to the tweet\_corp: clean\_corp

clean\_corp <- clean\_corpus(tweet\_corp)

# Print out a cleaned up tweet

content(clean\_corp[[227]])

# Print out the same tweet in original form

tweets$text[227]

# 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 and get some Starbucks while building a document-term matrix from coffee tweets.

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. The tm package uses a "simple triplet matrix" class. However, it is often easier to manipulate and examine the object by re-classifying the DTM with as.matrix()

##### Instructions

**100 XP**

* Create coffee\_dtm by applying DocumentTermMatrix() to clean\_corp.
* Create coffee\_m, a matrix version of coffee\_dtm, using as.matrix().
* Print the dimensions of coffee\_m to the console using the dim() function. Note the number of rows and columns.
* Print the subset of coffee\_m containing documents (rows) 25 through 35 and terms (columns) "star" and "starbucks".

[**Take Hint (-30 XP)**](javascript:void(0))

# Create the document-term matrix from the corpus

coffee\_dtm <- DocumentTermMatrix(clean\_corp)

# Print out coffee\_dtm data

coffee\_dtm

# Convert coffee\_dtm to a matrix

coffee\_m <- as.matrix(coffee\_dtm)

# Print the dimensions of coffee\_m

dim(coffee\_m)

# Review a portion of the matrix to get some Starbucks

coffee\_m[25:35, c("star", "starbucks")]

# 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.

##### Instructions

**100 XP**

* Create coffee\_tdm by applying TermDocumentMatrix() to clean\_corp.
* Create coffee\_m by converting coffee\_tdm to a matrix using as.matrix().
* Print the dimensions of coffee\_m to the console. Note the number of rows and columns.
* Print the subset of coffee\_m containing terms (rows) "star" and "starbucks" and documents (columns) 25 through 35.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create a term-document matrix from the corpus

coffee\_tdm <- TermDocumentMatrix(clean\_corp)

# Print coffee\_tdm data

coffee\_tdm

# Convert coffee\_tdm to a matrix

coffee\_m <- as.matrix(coffee\_tdm)

# Print the dimensions of the matrix

dim(coffee\_m)

# Review a portion of the matrix

coffee\_m[c("star", "starbucks"), 25:35]

# 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... :)

##### Instructions

**100 XP**

* Create coffee\_m as a matrix using the term-document matrix coffee\_tdm from the last chapter.
* Create term\_frequency using the rowSums() function on coffee\_m.
* Sort term\_frequency in descending order and store the result in term\_frequency.
* Use single square bracket subsetting, i.e. using only one [, to print the top 10 terms from term\_frequency.
* Make a barplot of the **top 10** terms.

[**Take Hint (-30 XP)**](javascript:void(0))

## coffee\_tdm is still loaded in your workspace

# Convert coffee\_tdm to a matrix

coffee\_m <- as.matrix(coffee\_tdm)

# Calculate the row sums of coffee\_m

term\_frequency <- rowSums(coffee\_m)

# Sort term\_frequency in decreasing order

term\_frequency <- sort(term\_frequency, decreasing = TRUE)

# View the top 10 most common words

term\_frequency[1:10]

# 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 preprocessing 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.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Create frequency using the freq\_terms() function on tweets$text. Include arguments to accomplish the following:
    - Limit to the top 10 terms.
    - At least 3 letters per term.
    - Use "Top200Words" to define stop words.
  + Produce a plot() of the frequency object. Compare it to the plot you produced in the previous exercise.

[**Take Hint (-15 XP)**](javascript:void(0))

* [2](javascript:void(0))
  + Again, create frequency using the freq\_terms() function on tweets$text. Include the following arguments:
    - Limit to the top 10 terms.
    - At least 3 letters per term.
    - This time use stopwords("english") to define stop words.
  + Produce a plot() of frequency. Compare it to the plot of frequency. Do certain words change based on the stop words criterion?

# Create frequency

frequency <- freq\_terms(tweets$text

,

top = 10,

at.least = 3,

stopwords = "Top200Words"

)

# Make a frequency barchart

plot(frequency)

# Create frequency

frequency <- freq\_terms(tweets$text

,

top = 10,

at.least = 3,

stopwords = stopwords("english")

)

# Make a frequency barchart

plot(frequency)

# 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 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.

##### Instructions

**100 XP**

* Load the wordcloud package.
* Print out first 10 entries in term\_frequency.
* Extract the terms using names() on term\_frequency. Call the vector of strings terms\_vec.
* Create a wordcloud() using terms\_vec as the words, and term\_frequency as the values. Add the parameters max.words = 50 and colors = "red".

[**Take Hint (-30 XP)**](javascript:void(0))

# Load wordcloud package

library(wordcloud)

# Print the first 10 entries in term\_frequency

term\_frequency[1:10]

# Vector of terms

terms\_vec<-names(term\_frequency)

# Create a wordcloud for the values in word\_freqs

wordcloud(terms\_vec,term\_frequency,max.words=50,colors="red")

# Stop words and word clouds

Now that you are in the text mining mindset, sitting down for a nice glass of chardonnay, we need to dig deeper. In the last word cloud, "chardonnay" dominated the visual. It was so dominant that you couldn't draw out any other interesting insights.

Let's change the stop words to include "chardonnay" to see what other words are common, yet were originally drowned out.

Your workspace has a cleaned version of chardonnay tweets but now lets remove some non-insightful terms. This exercise uses content() to show you a specific tweet for comparison. Remember to use double brackets to index the corpus list.

##### Instructions

**100 XP**

* Apply content() to the 24th document in chardonnay\_corp.
* Append "chardonnay" to the English stopwords, assigning to stops.
* Examine the last 6 words in stops.
* Create cleaned\_chardonnay\_corp with tm\_map() by passing in the chardonnay\_corp, the function removeWords and finally the stopwords, stops.
* Now examine the content of the 24 tweet again to compare results.

[**Take Hint (-30 XP)**](javascript:void(0))

# Review a "cleaned" tweet

content(chardonnay\_corp[[24]])

# Add to stopwords

stops <- c(stopwords(kind = 'en'), 'chardonnay')

# Review last 6 stopwords

tail(stops)

# Apply to a corpus

cleaned\_chardonnay\_corp <- tm\_map(chardonnay\_corp, removeWords, stops)

# Review a "cleaned" tweet again

content(cleaned\_chardonnay\_corp[[24]])

# Plot the better word cloud

Now that you've removed additional stopwords, let's take a look at the improved word cloud!

The term-document matrix from the previous exercise has been turned into matrix with as.matrix(), then a named vector was created with rowSums(). This new object of term frequencies called chardonnay\_words is preloaded into your workspace. Let's take a look at these new wordcloud results.

##### Instructions

**100 XP**

We've loaded the wordcloud package for you behind the scenes and will do so for all additional exercises requiring it.

* Sort the values in chardonnay\_words with decreasing = TRUE. Save as sorted\_chardonnay\_words.
* Look at the top 6 words in sorted\_chardonnay\_words and their values.
* Create terms\_vec using names() on chardonnay\_words.
* Pass terms\_vec and chardonnay\_words into the wordcloud() function. Review what other words pop out now that "chardonnay" is removed.

[**Take Hint (-30 XP)**](javascript:void(0))

# Sort the chardonnay\_words in descending order

sorted\_chardonnay\_words <- sort(chardonnay\_words, decreasing = TRUE)

# Print the 6 most frequent chardonnay terms

head(sorted\_chardonnay\_words)

# Get a terms vector

terms\_vec <- names(chardonnay\_words)

# Create a wordcloud for the values in word\_freqs

wordcloud(terms\_vec, chardonnay\_words,

max.words = 50, colors = "red")

# Improve word cloud colors

So far, you have specified only a single hexadecimal 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**](http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf) as a reference.

In this exercise you will use "grey80", "darkgoldenrod1", and "tomato" as colors. This is a good starting palette to highlight terms because "tomato" stands out more than "grey80". It is a best practice to start with three colors, each with increasing vibrancy. Doing so will naturally divide the term frequency into "low", "medium" and "high" for easier viewing.

##### Instructions

**100 XP**

* Call thecolors() function to list all basic colors.
* Create a wordcloud() using the predefined chardonnay\_freqs with the colors "grey80", "darkgoldenrod1", and "tomato". Include the top 100 terms using max.words.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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. So if you find yourself unable to pick colors on your own, you can use the viridisLite package. viridisLite color schemes are perceptually-uniform, both in regular form and when converted to black-and-white. The colors are also designed to be perceived by readers with color blindness.

There are multiple color palettes each with a convenience function. Simply specify n to select the number of colors needed.

magma(n = 3)

plasma(n = 5)

inferno(n = 6)

Each function returns a vector of hexadecimal colors based on n. Here's an example used with wordcloud:

color\_pal <- cividis(n = 7)

wordcloud(chardonnay\_freqs$term, chardonnay\_freqs$num, max.words = 100, colors = color\_pal)

##### Instructions

**100 XP**

* Use cividis() to select 5 colors in an object called color\_pal.
* Review the hexadecimal colors by printing color\_pal to your console.
* Create a wordcloud() from the chardonnay\_freqs term and num columns. Include the top 100 terms using max.words, and set the colors to your palette, color\_pal.

[**Take Hint (-30 XP)**](javascript:void(0))

# Select 5 colors

color\_pal <- cividis(5)

# Examine the palette output

color\_pal

# Create a wordcloud with the selected palette

wordcloud(chardonnay\_freqs$term, chardonnay\_freqs$num, max.words = 100, colors = color\_pal)

# 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 single vector containing the two collapsed documents.

a\_single\_string <- paste(a\_character\_vector, collapse = " ")

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.

##### Instructions

**100 XP**

* Create all\_coffee by using paste() with collapse = " " on coffee\_tweets$text.
* Create all\_chardonnay by using paste() with collapse = " " on chardonnay\_tweets$text.
* Create all\_tweets using c() to combine all\_coffee and all\_chardonnay. Make all\_coffee the first term.
* Convert all\_tweets using VectorSource().
* Create all\_corpus by using VCorpus() on all\_tweets.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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\_source<-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")

##### Instructions

**100 XP**

* Create all\_clean by applying the predefined clean\_corpus() function to all\_corpus.
* Create all\_tdm, a TermDocumentMatrix from all\_clean.
* Create all\_m by converting all\_tdm to a matrix object.
* Create a commonality.cloud() from all\_m with max.words = 100 and colors = "steelblue1".

[**Take Hint (-30 XP)**](javascript:void(0))

# 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.

##### Instructions

**100 XP**

all\_corpus is preloaded in your workspace.

* Create all\_clean by applying the predefined clean\_corpus function to all\_corpus.
* Create all\_tdm, a TermDocumentMatrix, from all\_clean.
* Use colnames() to rename each distinct corpora within all\_tdm. Name the first column "coffee" and the second column "chardonnay".
* Create all\_m by converting all\_tdm into matrix form.
* Create a comparison.cloud() using all\_m, with colors = c("orange", "blue") and max.words = 50.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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, colors = c("orange", "blue"), max.words = 50)

# Polarized tag cloud

Commonality clouds show words that are shared across documents. One interesting thing that they can't show you is which of those words appear more commonly in one document compared to another. For this, you need a pyramid plot; these can be generated using pyramid.plot() from the plotrix package.

First, some manipulation is required to get the data in a suitable form. This is most easily done by converting it to a data frame and using dplyr. Given a matrix of word counts, as created by as.matrix(tdm), you need to end up with a data frame with three columns:

* The words contained in each document.
* The counts of those words from document 1.
* The counts of those words from document 2.

Then pyramid.plot() using

pyramid.plot(word\_count\_data$count1, word\_count\_data$count2, word\_count\_data$word)

There are some additional arguments to improve the cosmetic appearance of the plot.

Now you'll explore words that are common in chardonnay tweets, but rare in coffee tweets. all\_dtm\_m is created for you.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Convert all\_tdm\_m to a data frame. Set the rownames to a column named "word".
* Filter to keep rows where all columns are greater than zero, using the syntax . > 0.
* Add a column named difference, equal to the count in the chardonnay column minus the count in the coffee column.
* Filter to keep the top 25 rows by difference.
* Arrange the rows by desc()ending order of difference.

[**Take Hint (-15 XP)**](javascript:void(0))

* Set the left count to the chardonnay column.
* Set the right count to the coffee column.
* Set the labels to the word column.

[**Take Hint (-15 XP)**](javascript:void(0))

top25\_df <- all\_tdm\_m %>%

# Convert to data frame

as\_data\_frame(rownames = "word") %>%

# Keep rows where word appears everywhere

filter\_all(all\_vars(. > 0)) %>%

# Get difference in counts

mutate(difference = chardonnay - coffee) %>%

# Keep rows with biggest difference

top\_n(25, wt = difference) %>%

# Arrange by descending difference

arrange(desc(difference))

pyramid.plot(

# Chardonnay counts

top25\_df$chardonnay,

# Coffee counts

top25\_df$coffee,

# Words

labels = top25\_df$word,

top.labels = c("Chardonnay", "Words", "Coffee"),

main = "Words in Common",

unit = NULL,

gap = 8,

)

# Visualize word networks

Another way to view word connections is to treat them as a network, similar to a social network. Word networks show term association and cohesion. A word of caution: these visuals can become very dense and hard to interpret visually.

In a network graph, the circles are called nodes and represent individual terms, while the lines connecting the circles are called edges and represent the connections between the terms.

For the over-caffeinated text miner, qdap provides a shorcut for making word networks. The word\_network\_plot() and word\_associate() functions both make word networks easy!

The sample code constructs a word network for words associated with "Marvin".

##### Instructions

**100 XP**

Update the word\_associate() plotting code to work with the coffee data.

* Change the vector to coffee\_tweets$text.
* Change the match string to "barista".
* Change "chardonnay" to "coffee" in the stopwords too.
* Change the title to "Barista Coffee Tweet Associations" in the sample code for the plot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Word association

word\_associate(coffee\_tweets$text, match.string = "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

In the next chapter, we cover some miscellaneous (yet very important) text mining subjects including:

* TDM/DTM weighting
* Dealing with TDM/DTM sparsity
* Capturing metadata
* Simple word clustering for topics
* Analysis on more than one word

For now, let's simply create a new visual called a dendrogram from our coffee\_tweets. The next chapter will explain it in detail.

##### Instructions

**100 XP**

A hierarchical cluster object, hc, has been created for you from the coffee tweets.

Create a dendrogram using plot() on hc.

[**Take Hint (-30 XP)**](javascript:void(0))

# Plot a dendrogram

plot(hc)

# 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

##### Instructions

**100 XP**

The data frame rain has been preloaded in your workspace.

* Create dist\_rain by using the dist() function on the values in the second column of rain.
* Print the dist\_rain matrix to the console.
* Create hc by performing a cluster analysis, using hclust() on dist\_rain.
* plot() the hc object with labels = rain$city to add the city names.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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 dendrogram friendly 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.

In most professional settings a good dendrogram is based on a TDM with 25 to 70 terms. Having more than 70 terms may mean the visual will be cluttered and incomprehensible. Conversely, having less than 25 terms likely means your dendrogram may not plot relevant and insightful clusters.

When using removeSparseTerms(), the sparse parameter will adjust the total terms kept in the TDM. The closer sparse is to 1, the more terms are kept. This value represents a percentage cutoff of zeros for each term in the TDM.

##### Instructions

**100 XP**

tweets\_tdm has been created using the chardonnay tweets.

* Print the dimensions of tweets\_tdm to the console.
* Create tdm1 using removeSparseTerms() with sparse = 0.95 on tweets\_tdm.
* Create tdm2 using removeSparseTerms() with sparse = 0.975 on tweets\_tdm.
* Print tdm1 to the console to see how many terms are left.
* Print tdm2 to the console to see how many terms are left.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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()), 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.

##### Instructions

**100 XP**

* Create tweets\_tdm2 by applying removeSparseTerms() on tweets\_tdm. Use sparse = 0.975.
* Create tdm\_m by using as.matrix() on tweets\_tdm2 to convert it to matrix form.
* Create tweets\_dist containing the distances of tdm\_m using the dist() function.
* Create a hierarchical cluster object called hc using hclust() on tweets\_dist.
* Make a dendrogram with plot() and hc.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create tweets\_tdm2

tweets\_tdm2 <- removeSparseTerms(tweets\_tdm, sparse=0.975)

# Create tdm\_m

tdm\_m <- as.matrix(tweets\_tdm2)

# Create tweets\_dist

tweets\_dist <- dist(tdm\_m)

# Create hc

hc <- hclust(tweets\_dist)

# Plot the dendrogram

plot(hc)

# Dendrogram aesthetics

So you made a dendrogram...but it's 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").

##### Instructions

**100 XP**

The dendextend package has been loaded for you, and a hierarchical cluster object, hc, was created from tweets\_dist.

* Create hcd as a dendrogram using as.dendrogram() on hc.
* Print the labels of hcd to the console.
* Use branches\_attr\_by\_labels() to color the branches. Pass it three arguments: the hcd object, c("marvin", "gaye"), and the color "red". Assign to hcd\_colored.
* plot() the dendrogram hcd\_colored with the title "Better Dendrogram", added using the main argument.
* Add rectangles to the plot using rect.dendrogram(). Specify k = 2 clusters and a border color of "grey50".

[**Take Hint (-30 XP)**](javascript:void(0))

# Create hcd

hcd <- as.dendrogram(hc)

# Print the labels in hcd

labels(hcd)

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

hcd\_colored <- branches\_attr\_by\_labels(hcd, c("marvin","gaye"), color="red")

# Plot hcd

plot(hcd\_colored, main="Better Dendrogram")

# Add cluster rectangles

rect.dendrogram(hcd\_colored, 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 in documents, while a score approaching 0 means the terms seldom appear in the same document.

Keep in mind the calculation for findAssocs() is done at the document level. So for every document that contains the word in question, the other terms in those specific documents are associated. Documents without the search term are ignored.

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.

##### Instructions

**100 XP**

* Create associations using findAssocs() on tweets\_tdm to find terms associated with "venti", which meet a minimum threshold of 0.2.
* View the terms associated with "venti" by printing associations to the console.
* Create associations\_df, by calling list\_vect2df(), passing associations, then setting col2 to "word" and col3 to "score".
* Run the ggplot2 code to make a dot plot of the association values.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create associations

associations <- findAssocs(tweets\_tdm, "venti", corlimit=0.2)

# View the venti associations

associations

# Create associations\_df

associations\_df <- list\_vect2df(associations, col2="word", col3="score")

# Plot the associations\_df values

ggplot(associations\_df, aes(score, word)) +

geom\_point(size = 3) +

theme\_gdocs()

# 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)

)

##### Instructions

**100 XP**

A corpus has been preprocessed as before using the chardonnay tweets. The resulting object text\_corp is available in your workspace.

* Create a tokenizer function like the above which creates 2-word bigrams.
* Make unigram\_dtm by calling DocumentTermMatrix() on text\_corp without using the tokenizer() function.
* Make bigram\_dtm using DocumentTermMatrix() on text\_corp with the tokenizer() function you just made.
* Examine unigram\_dtm and bigram\_dtm. Which has more terms?

[**Take Hint (-30 XP)**](javascript:void(0))

# 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)

)

# Print unigram\_dtm

unigram\_dtm

# Print 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 terms in the chardonnay word cloud? Using bigram tokenization grabs all two word combinations. Observe what happens to the word cloud in this exercise.

This exercise uses str\_subset from stringr. Keep in mind, other DataCamp courses cover regular expressions in more detail. As a reminder, the regular expression ^ matches the starting position within the exercise's bigrams.

##### Instructions

**100 XP**

The chardonnay tweets have been cleaned and organized into a DTM called bigram\_dtm.

* Create bigram\_dtm\_m by converting bigram\_dtm to a matrix.
* Create an object freq consisting of the word frequencies by applying colSums() on bigram\_dtm\_m.
* Extract the character vector of word combinations with names(freq) and assign the result to bi\_words.
* Pass bi\_words to str\_subset() with the matching pattern "^marvin" to review all bigrams starting with "marvin".
* Plot a simple wordcloud() passing bi\_words, freq and max.words = 15 into the function.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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

str\_subset(bi\_words, "^marvin")

# Plot a wordcloud

wordcloud(bi\_words, freq, max.words=15)

# Changing frequency weights

So far you've simply counted terms in documents in the DocumentTermMatrix or TermDocumentMatrix. In this exercise, you'll learn about TfIdf weighting instead of simple term frequency. TfIdf stands for term frequency-inverse document frequency and is used when you have a large corpora with limited term diversity.

TfIdf counts terms (i.e. Tf), normalizes the value by document length, and then penalizes the value the more often a word appears among the documents. This is common sense; if a word is commonplace it's important but not insightful. This penalty aspect is captured in the inverse document frequency (i.e. Idf).

For example, reviewing customer service notes may include the term "cu" as shorthand for customer. One note may state "the cu has a damaged package" and another as "cu called with question about delivery". With document frequency weighting, "cu" appears twice so it is expected to be informative. However in TfIdf, "cu" is penalized because it appears in all the documents. As a result "cu" isn't considered novel so its value is reduced towards 0 which lets other terms have higher values for analysis.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Create tdm, a term frequency-based TermDocumentMatrix() using text\_corp.
  + Create tdm\_m by converting tdm to matrix form.
  + Examine the term frequency for "coffee", "espresso", and "latte" in a few tweets. Subset tdm\_m to get rows c("coffee", "espresso", "latte") and columns 161 to 166.

[**Take Hint (-15 XP)**](javascript:void(0))

* [2](javascript:void(0))
  + Edit the TermDocumentMatrix() to use TfIdf weighting. Pass control = list(weighting = weightTfIdf) as an argument to the function.
  + Run the code and compare the new scores to the first part of the exercise.

# Edit the controls to use Tfidf weighting

tdm <- TermDocumentMatrix(text\_corp,control=list(weighting=weightTfIdf))

# Convert to matrix again

tdm\_m <- as.matrix(tdm)

# Examine the same part: how has it changed?

tdm\_m[c("coffee", "espresso", "latte"), 161:166]

# 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 corpus.

To capture document level metadata, the column names and order **must** be:

1. doc\_id - a unique string for each document
2. text - the text to be examined
3. ... - any other columns will be automatically catalogued as metadata.

Sometimes you will need to rename columns in order to fit the expectations of DataframeSource(). The names() function is helpful for this.

tweets exists in your worksapce as a data frame with columns "num", "text", "screenName", and "created".

##### Instructions

**100 XP**

* Rename the first column of tweets to "doc\_id".
* Set the document schema with DataframeSource() on the smaller tweets data frame.
* Make the document collection a volatile corpus nested in the custom clean\_corpus() function.
* Apply content() to the first tweet with double brackets such as text\_corpus[[1]] to see the cleaned plain text.
* Confirm that all metadata was captured using the [**meta()**](https://www.rdocumentation.org/packages/tm/topics/meta) function on the first document with single brackets.

Remember, when accessing part of a corpus the double or single brackets make a difference! For this exercise you will use double brackets with content() and single brackets with meta().

[**Take Hint (-30 XP)**](javascript:void(0))

# Rename columns

names(tweets)[1] <- "doc\_id"

# Set the schema: docs

docs <- DataframeSource(tweets)

# Make a clean volatile corpus: text\_corpus

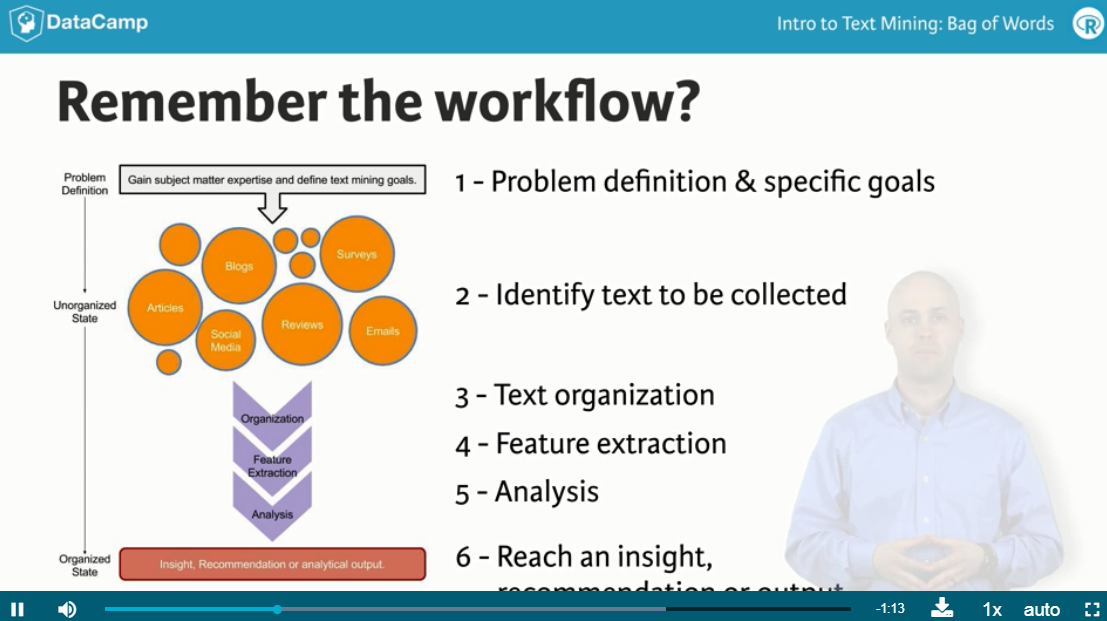
text\_corpus <- clean\_corpus(VCorpus(docs))

# Examine the first doc content

content(text\_corpus[[1]])

# Access the first doc metadata

meta(text\_corpus[1])



# 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**](http://www.indeed.com/cmp/Amazon.com/reviews), [**Glassdoor**](http://www.glassdoor.com/) or [**CareerBliss**](http://www.careerbliss.com/amazon/reviews/).

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

##### Instructions

**100 XP**

* View the structure of amzn with str() to get its dimensions and a preview of the data.
* Create amzn\_pros from the positive reviews column amzn$pros.
* Create amzn\_cons from the negative reviews column amzn$cons.
* Print the structure of goog with str() to get its dimensions and a preview of the data.
* Create goog\_pros from the positive reviews column goog$pros.
* Create goog\_cons from the negative reviews column goog$cons.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Apply qdap\_clean() to amzn\_pros, assigning to qdap\_cleaned\_amzn\_pros.
  + Create a vector source (VectorSource()) from qdap\_cleaned\_amzn\_pros, then turn it into a volatile corpus (VCorpus()), assigning to amzn\_p\_corp.
  + Create amzn\_pros\_corp by applying tm\_clean() to amzn\_p\_corp.

[**Take Hint (-15 XP)**](javascript:void(0))

* [2](javascript:void(0))
  + Apply qdap\_clean() to amzn\_cons, assigning to qdap\_cleaned\_amzn\_cons.
  + Create a vector source from qdap\_cleaned\_amzn\_cons, then turn it into a volatile corpus, assigning to amzn\_c\_corp.
  + Create amzn\_cons\_corp by applying tm\_clean() to amzn\_c\_corp.

# qdap\_clean the text

qdap\_cleaned\_amzn\_pros <- qdap\_clean(amzn\_pros)

# Source and create the corpus

amzn\_p\_corp <- VCorpus(VectorSource(qdap\_cleaned\_amzn\_pros))

# tm\_clean the corpus

amzn\_pros\_corp <- tm\_clean(amzn\_p\_corp)

# qdap\_clean the text

qdap\_cleaned\_amzn\_cons <- qdap\_clean(amzn\_cons)

# Source and create the corpus

amzn\_c\_corp <- VCorpus(VectorSource(qdap\_cleaned\_amzn\_cons))

# tm\_clean the corpus

amzn\_cons\_corp <- tm\_clean(amzn\_c\_corp)

# Working with Google reviews

Now that the Amazon reviews have been cleaned, the same must be done for the Google reviews. qdap\_clean() and tm\_clean() are available in your workspace to help you clean goog\_pros and goog\_cons.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Apply qdap\_clean() to goog\_pros, assigning to qdap\_cleaned\_goog\_pros.
  + Create a vector source (VectorSource()) from qdap\_cleaned\_goog\_pros, then turn it into a volatile corpus (VCorpus()), assigning to goog\_p\_corp.
  + Create goog\_pros\_corp by applying tm\_clean() to goog\_p\_corp.

[**Take Hint (-15 XP)**](javascript:void(0))

* [2](javascript:void(0))
  + Apply qdap\_clean() to goog\_cons, assigning to qdap\_cleaned\_goog\_cons.
  + Create a vector source from qdap\_cleaned\_goog\_cons, then turn it into a volatile corpus, assigning to goog\_c\_corp.
  + Create goog\_cons\_corp by applying tm\_clean() to goog\_c\_corp.

# qdap\_clean the text

qdap\_cleaned\_goog\_pros <- qdap\_clean(goog\_pros)

# Source and create the corpus

goog\_p\_corp <- VCorpus(VectorSource(qdap\_cleaned\_goog\_pros))

# tm\_clean the corpus

goog\_pros\_corp <- tm\_clean(goog\_p\_corp)

# qdap clean the text

qdap\_cleaned

# Source and create the corpus

goog\_c\_corp <- VCorpus(VectorSource(qdap\_cleaned\_goog\_cons))

# tm clean the corpus

goog\_cons\_corp <- tm\_clean(goog\_c\_corp)

# 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))

}

##### Instructions

**100 XP**

* Create amzn\_p\_tdm as a TermDocumentMatrix from amzn\_pros\_corp. Make sure to add control = list(tokenize = tokenizer) so that the terms are bigrams.
* Create amzn\_p\_tdm\_m from amzn\_p\_tdm by using the as.matrix() function.
* Create amzn\_p\_freq to obtain the term frequencies from amzn\_p\_tdm\_m.
* Create a wordcloud() using names(amzn\_p\_freq) as the words, amzn\_p\_freq as their frequencies, and max.words = 25 and color = "blue" for aesthetics.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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, color = "blue")

# Feature extraction & analysis: amzn\_cons

You now decide to contrast this with the amzn\_cons\_corp corpus in another bigram TDM. Of course, you expect to see some different phrases in your word cloud.

Once again, you will use this custom function to extract your bigram features for the visual:

tokenizer <- function(x)

NGramTokenizer(x, Weka\_control(min = 2, max = 2))

##### Instructions

**100 XP**

* Create amzn\_c\_tdm by converting amzn\_cons\_corp into a TermDocumentMatrix and incorporating the bigram function control = list(tokenize = tokenizer).
* Create amzn\_c\_tdm\_m as a matrix version of amzn\_c\_tdm.
* Create amzn\_c\_freq by using rowSums() to get term frequencies from amzn\_c\_tdm\_m.
* Create a wordcloud() using names(amzn\_c\_freq) and the values amzn\_c\_freq. Use the arguments max.words = 25 and color = "red" as well.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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.

##### Instructions

**100 XP**

* Create amzn\_c\_tdm as a TermDocumentMatrix using amzn\_cons\_corp with control = list(tokenize = tokenizer).
* Print amzn\_c\_tdm to the console.
* Create amzn\_c\_tdm2 by applying the removeSparseTerms() function to amzn\_c\_tdm with the sparse argument equal to .993.
* Create hc, a hierarchical cluster object by nesting the distance matrix dist(amzn\_c\_tdm2) inside the hclust() function. Make sure to also pass method = "complete" to the hclust() function.
* Plot hc to view the clustered bigrams and see how the concepts in the Amazon cons section may lead you to a conclusion.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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, .993)

# Create hc as a cluster of distance values

hc <- hclust(dist(amzn\_c\_tdm2),

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.

##### Instructions

**0 XP**

##### Instructions

**0 XP**

The amzn\_pros\_corp corpus has been cleaned using the custom functions like before.

* Construct a TDM called amzn\_p\_tdm from amzn\_pros\_corp and control = list(tokenize = tokenizer).
* Create amzn\_p\_m by converting amzn\_p\_tdm to a matrix.
* Create amzn\_p\_freq by applying rowSums() to amzn\_p\_m.
* Create term\_frequency using sort() on amzn\_p\_freq along with the argument decreasing = TRUE.
* Examine the first 5 bigrams using term\_frequency[1:5].
* You may be surprised to see "fast paced" as a top term because it could be a negative term related to "long hours". Look at the terms most associated with "fast paced". Use findAssocs() on amzn\_p\_tdm to examine "fast paced" with a 0.2 cutoff.

#

#

# 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

term\_frequency[1:5]

# 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.

##### Instructions

**100 XP**

The all\_goog\_corpus object consisting of Google pro and con reviews is loaded in your workspace.

* Create all\_goog\_corp by cleaning all\_goog\_corpus with the predefined tm\_clean() function.
* Create all\_tdm by converting all\_goog\_corp to a term-document matrix.
* Create all\_m by converting all\_tdm to a matrix.
* Construct a comparison.cloud() from all\_m. Set max.words to 100. The colors argument is specified for you.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create all\_goog\_corp

all\_goog\_corp <- tm\_clean(all\_goog\_corpus)

# Create all\_tdm

all\_tdm <- TermDocumentMatrix(all\_goog\_corp)

# Create all\_m

all\_m <- as.matrix(all\_tdm)

# Build a comparison cloud

comparison.cloud(all\_m,

max.words = 100, colors = c("#F44336", "#2196f3"))

# 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 compare the differences between any shared bigrams.  
We have preloaded a data frame, all\_tdm\_df, consisting of terms and corresponding AmazonPro, and GooglePro bigram frequencies. Using this data frame you will identify the top 5 bigrams that are shared between the two corpora.

##### Instructions

**100 XP**

* Create common\_words from all\_tdm\_df using dplyr functions.
  + filter() on the AmazonPro column for nonzero values.
  + Likewise filter the GooglePro column for nonzero values.
  + Then mutate() a new column, diff which is the abs (absolute) difference between the term frequencies columns.
* Although we could have piped again, create top5\_df by applying top\_n to common\_words to extract the top 5 values in the diff column. It will print to your console for review.
* Create a pyramid.plot passing in top5\_df$AmazonPro then top5\_df$GooglePro and finally add labels with top5\_df$terms.

[**Take Hint (-30 XP)**](javascript:void(0))

# Filter to words in common and create an absolute diff column

common\_words <- all\_tdm\_df %>%

filter(

AmazonPro != 0,

GooglePro != 0

) %>%

mutate(diff = abs(AmazonPro - GooglePro))

# Extract top 5 common bigrams

(top5\_df <- top\_n(common\_words, 5, diff))

# Create the pyramid plot

pyramid.plot(top5\_df$AmazonPro, top5\_df$GooglePro,

labels = top5\_df$terms, gap = 12,

top.labels = c("Amzn", "Pro Words", "Goog"),

main = "Words in Common", unit = NULL)

# Cage match, part 2! Negative reviews

In both organizations, people mentioned "culture" and "smart people", so there are some similar positive aspects between the two companies. However with the pyramid plot you can start to infer degrees of positive features of the work environments.

You now decide to turn your attention to negative reviews and make the same visual. This time you already have the common\_words data frame in your workspace. However, the common bigrams in this exercise come form negative employee reviews.

##### Instructions

**100 XP**

* Using top\_n() on common\_words, obtain the top 5 bigrams weighted on the diff column. The results of the new object will print to your console.
* Create a pyramid.plot(). Pass in top5\_df$AmazonNeg, top5\_df$GoogleNeg, and labels = top5\_df$terms. For better labelling, set
  + gap to 12.
  + top.labels to c("Amzn", "Neg Words", "Goog")

The main and unit arguments are set for you.

[**Take Hint (-30 XP)**](javascript:void(0))

# Extract top 5 common bigrams

(top5\_df <- top\_n(common\_words, 5, diff))

# Create a pyramid plot

pyramid.plot(

# Amazon on the left

top5\_df$AmazonNeg,

# Google on the right

top5\_df$GoogleNeg,

# Use terms for labels

labels = top5\_df$terms,

# Set the gap to 12

gap = 12,

# Set top.labels to "Amzn", "Neg words" & "Goog"

top.labels =c("Amzn","Neg Words","Goog"),

main = "Words in Common",

unit = NULL

)