

CHAPTER 1

The Tidy Text Format

Using tidy data principles is a powerful way to make handling data easier and more effective, and this is no less true when it comes to dealing with text. As described by Hadley Wickham (Wickham 2014), tidy data has a specific structure:

- Each variable is a column.
- Each observation is a row.
- Each type of observational unit is a table.

We thus define the tidy text format as being *a table with one token per row*. A token is a meaningful unit of text, such as a word, that we are interested in using for analysis, and tokenization is the process of splitting text into tokens. This one-token-per-row structure is in contrast to the ways text is often stored in current analyses, perhaps as strings or in a document-term matrix. For tidy text mining, the *token* that is stored in each row is most often a single word, but can also be an n-gram, sentence, or paragraph. In the tidytext package, we provide functionality to tokenize by commonly used units of text like these and convert to a one-term-per-row format.

Tidy data sets allow manipulation with a standard set of “tidy” tools, including popular packages such as dplyr (Wickham and Francois 2016), tidyr (Wickham 2016), ggplot2 (Wickham 2009), and broom (Robinson 2017). By keeping the input and output in tidy tables, users can transition fluidly between these packages. We’ve found these tidy tools extend naturally to many text analyses and explorations.

At the same time, the tidytext package doesn’t expect a user to keep text data in a tidy form at all times during an analysis. The package includes functions to tidy() objects (see the broom package [Robinson, cited above]) from popular text mining R packages such as tm (Feinerer et al. 2008) and quanteda (Benoit and Nulty 2016). This allows, for example, a workflow where importing, filtering, and processing is done

not tidy
↑

using dplyr and other tidy tools, after which the data is converted into a document-term matrix for machine learning applications. The models can then be reconverted into a tidy form for interpretation and visualization with ggplot2.

tidy with dplyr → dfm → tidy

Contrasting Tidy Text with Other Data Structures for ggplot

As we stated above, we define the tidy text format as being a table with *one token per row*. Structuring text data in this way means that it conforms to tidy data principles and can be manipulated with a set of consistent tools. This is worth contrasting with the ways text is often stored in text mining approaches:

String

Text can, of course, be stored as strings (i.e., character vectors) within R, and often text data is first read into memory in this form.

this is
a very
Jockers
approach

Corpus

These types of objects typically contain raw strings annotated with additional metadata and details.

Document-term matrix

This is a sparse matrix describing a collection (i.e., a corpus) of documents with one row for each document and one column for each term. The value in the matrix is typically word count or tf-idf (see Chapter 3).

Let's hold off on exploring corpus and document-term matrix objects until Chapter 5, and get down to the basics of converting text to a tidy format.

The unnest_tokens Function

File "chapter 1"

Emily Dickinson wrote some lovely text in her time.

```
✓ text <- c("Because I could not stop for Death -",
      "He kindly stopped for me -",
      "The Carriage held but just Ourselves -",
      "and Immortality")

text
✓ ## [1] "Because I could not stop for Death -" "He kindly stopped for me -"
✓ ## [3] "The Carriage held but just Ourselves -" "and Immortality"
```

This is a typical character vector that we might want to analyze. In order to turn it into a tidy text dataset, we first need to put it into a data frame.

```
library(dplyr)
text_df <- data_frame(line = 1:4, text = text)

text_df
```

```
## # A tibble: 4 × 2
##   line      text
##   <int>    <chr>
## 1     1 Because I could not stop for Death -
## 2     2           He kindly stopped for me -
## 3     3 The Carriage held but just Ourselves -
## 4     4           and Immortality
```

- What does it mean that this data frame has printed out as a "tibble"? A *tibble* is a modern class of data frame within R, available in the dplyr and tibble packages, that ✓ has a convenient print method, will not convert strings to factors, and does not use row names. **Tibbles are great for use with tidy tools.**

- Notice that **this data frame containing text isn't yet compatible with tidy text analysis.** ✓ We can't filter out words or count which occur most frequently, since each row is made up of multiple combined words. **We need to convert this so that it has one token per document per row.** → ie "tokenize it".



A **token** is a meaningful unit of text, most often a word, that we are interested in using for further analysis, and tokenization is the process of splitting text into tokens.

In this first example, we only have one document (the poem), but we will explore examples with multiple documents soon. → **corpus**

Within our tidy text framework, **we need to both break the text into individual tokens (a process called tokenization) and transform it to a tidy data structure.** To do this, we use the **tidytext unnest_tokens()** function.

```
✓ library(tidytext)

✓ text_df %>%
  unnest_tokens(word, text)

## # A tibble: 20 × 2
##   line      word
##   <int>    <chr>
## 1     1 because
## 2     1 i
## 3     1 could
## 4     1 not
## 5     1 stop
## 6     1 for
## 7     1 death
## 8     2 he
## 9     2 kindly
## 10    2 stopped
## # ... with 10 more rows
```

untidy text → tokenize
+ tidy → data analysis

arg 1 arg 2
 \downarrow \downarrow
 unnest, tokens (word, text)

The two basic arguments to `unnest_tokens` used here are column names. First we have the output column name that will be created as the text is unnested into it (`word`, in this case), and then the input column that the text comes from (`text`, in this case). Remember that `text_df` above has a column called `text` that contains the data of interest.

After using `unnest_tokens`, we've split each row so that there is one token (word) in each row of the new data frame; the default tokenization in `unnest_tokens()` is for single words, as shown here. Also notice:

- Other columns, such as the line number each word came from, are retained.
- Punctuation has been stripped.
- By default, `unnest_tokens()` converts the tokens to lowercase, which makes them easier to compare or combine with other datasets. (Use the `to_lower = chapters?` `FALSE` argument to turn off this behavior).

Having the text data in this **tidy** format lets us manipulate, process, and visualize the text using the standard set of tidy tools, namely `dplyr`, `tidyr`, and `ggplot2`, as shown in Figure 1-1.

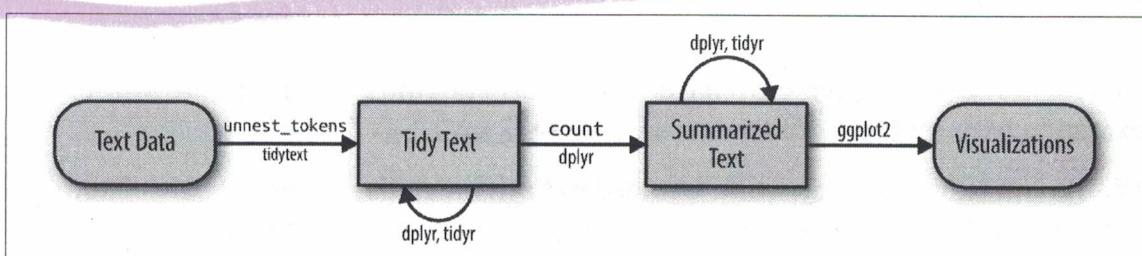


Figure 1-1. A flowchart of a typical text analysis using tidy data principles. This chapter shows how to summarize and visualize text using these tools.

Tidying the Works of Jane Austen (bigger corpus)

Let's use the text of Jane Austen's six completed, published novels from the `janeaustenr` package (Silge 2016), and transform them into a tidy format. The `janeaustenr` package provides these texts in a one-row-per-line format, where a line in this context is analogous to a literal printed line in a physical book. Let's start with that, and also use `mutate()` to annotate a `linenumber` quantity to keep track of lines in the original format, and a `chapter` (using a regex) to find where all the chapters are.

```

library(janeaustenr)
library(dplyr)
library(stringr)

original_books <- austen_books() %>%
  group_by(book) %>%
  
```

```

mutate(linenumber = row_number(),
       chapter = cumsum(str_detect(text, regex("^chapter [\\\[divxlc]", ignore_case = TRUE)))) %>%
ungroup()

original_books

## # A tibble: 73,422 × 4
##   text          book linenumber chapter
##   <chr>        <fctr>     <int>    <int>
## 1 SENSE AND SENSIBILITY Sense & Sensibility     1      0
## 2                               Sense & Sensibility     2      0
## 3           by Jane Austen Sense & Sensibility     3      0
## 4                               Sense & Sensibility     4      0
## 5           (1811) Sense & Sensibility     5      0
## 6                               Sense & Sensibility     6      0
## 7                               Sense & Sensibility     7      0
## 8                               Sense & Sensibility     8      0
## 9                               Sense & Sensibility     9      0
## 10          CHAPTER 1 Sense & Sensibility    10      1
## # ... with 73,412 more rows

```

To work with this as a tidy dataset, we need to restructure it in the one-token-per-row format, which as we saw earlier is done with the `unnest_tokens()` function.

```

✓ library(tidytext)
tidy_books <- original_books %>%
  unnest_tokens(word, text)

tidy_books

## # A tibble: 725,054 × 4
##   book linenumber chapter      word
##   <fctr>     <int>    <int>    <chr>
## 1 Sense & Sensibility     1      0    sense
## 2 Sense & Sensibility     1      0    and
## 3 Sense & Sensibility     1      0 sensibility
## 4 Sense & Sensibility     3      0      by
## 5 Sense & Sensibility     3      0      jane
## 6 Sense & Sensibility     3      0      austen
## 7 Sense & Sensibility     5      0      1811
## 8 Sense & Sensibility    10      1 chapter
## 9 Sense & Sensibility    10      1      1
## 10 Sense & Sensibility   13      1      the
## # ... with 725,044 more rows

```

This function uses the tokenizers package to separate each line of text in the original data frame into tokens. The default tokenizing is for words, but other options include characters, n-grams, sentences, lines, paragraphs, or separation around a regex pattern.

Now that the data is in one-word-per-row format, we can manipulate it with tidy tools like `dplyr`. Often in text analysis, we will want to remove "stop words," which are

words that are not useful for an analysis, typically extremely common words such as “the,” “of,” “to,” and so forth in English. We can remove stop words (kept in the tidytext dataset `stop_words`) with an `anti_join()`.

```
✓ data(stop_words)  
tidy_books <- tidy_books %>%  
anti_join(stop_words)
```

The `stop_words` dataset in the tidytext package contains stop words from three lexicons. We can use them all together, as we have here, or `filter()` to only use one set of stop words if that is more appropriate for a certain analysis.

We can also use dplyr’s `count()` to find the most common words in all the books as a whole.

```
tidy_books %>%  
count(word, sort = TRUE)  
  
## # A tibble: 13,914 × 2  
##   word     n  
##   <chr> <int>  
## 1 miss    1855  
## 2 time    1337  
## 3 fanny   862  
## 4 dear    822  
## 5 lady    817  
## 6 sir     806  
## 7 day     797  
## 8 emma    787  
## 9 sister   727  
## 10 house   699  
## # ... with 13,904 more rows
```

Because we’ve been using tidy tools, our word counts are stored in a tidy data frame. This allows us to pipe directly to the ggplot2 package, for example to create a visualization of the most common words (Figure 1-2).

```
library(ggplot2)  
  
tidy_books %>%  
count(word, sort = TRUE) %>%  
✓ filter(n > 600) %>%  
mutate(word = reorder(word, n)) %>%  
ggplot(aes(word, n)) +  
geom_col() +  
xlab(NULL) +  
coord_flip()
```

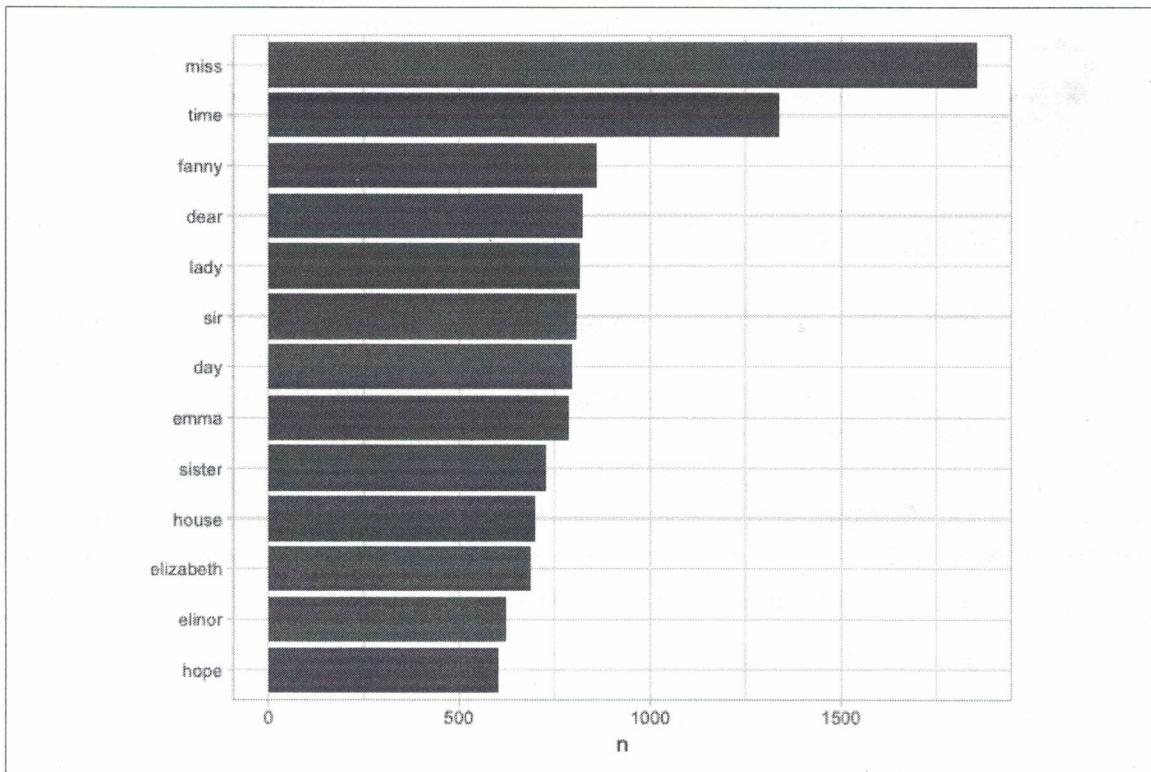


Figure 1-2. The most common words in Jane Austen’s novels

Note that the `austen_books()` function started us with exactly the text we wanted to analyze, but in other cases we may need to perform cleaning of text data, such as removing copyright headers or formatting. You’ll see examples of this kind of pre-processing in the case study chapters, particularly “Preprocessing” on page 153.

The gutenbergr Package

Now that we’ve used the `janeaustenr` package to explore tidying text, let’s introduce the `gutenbergr` package (Robinson 2016). The `gutenbergr` package provides access to the public domain works from the Project Gutenberg collection. The package includes tools both for downloading books (stripping out the unhelpful header/footer information), and a complete dataset of Project Gutenberg metadata that can be used to find works of interest. In this book, we will mostly use the `gutenberg_download()` function that downloads one or more works from Project Gutenberg by ID, but you can also use other functions to explore metadata, pair Gutenberg ID with title, author, language, and so on, or gather information about authors.



To learn more about gutenbergr, check out the package's tutorial at rOpenSci, where it is one of rOpenSci's packages for data access.

Word Frequencies

A common task in text mining is to **look at word frequencies**, just like we have done above for Jane Austen's novels, and to compare frequencies across different texts. We can do this intuitively and smoothly using tidy data principles. We already have Jane Austen's works; let's get two more sets of texts to compare to. First, let's look at some science fiction and fantasy novels by H.G. Wells, who lived in the late 19th and early 20th centuries. Let's get *The Time Machine*, *The War of the Worlds*, *The Invisible Man*, and *The Island of Doctor Moreau*. We can access these works using `gutenberg_download()` and the Project Gutenberg ID numbers for each novel.

How do we
find the
Project
Gutenberg
ID numbers
for each
novel?

What does
the "%>%"
actually do?

```
library(gutenberg)
hgwells <- gutenberg_download(c(35, 36, 5230, 159))
tidy_hgwells <- hgwells %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)    ✓
```

Just for kicks, what are the most common words in these novels of H.G. Wells?

```
tidy_hgwells %>%
  count(word, sort = TRUE)

## # A tibble: 11,769 × 2
##       word     n
##   <chr> <int>
## 1 time    454
## 2 people   302
## 3 door     260
## 4 heard    249
## 5 black    232
## 6 stood    229
## 7 white    222
## 8 hand     218
## 9 kemp     213
## 10 eyes    210
## # ... with 11,759 more rows
```

Now let's get some well-known works of the Brontë sisters, whose lives overlapped with Jane Austen's somewhat, but who wrote in a rather different style. Let's get *Jane Eyre*, *Wuthering Heights*, *The Tenant of Wildfell Hall*, *Villette*, and *Agnes Grey*. We will again use the Project Gutenberg ID numbers for each novel and access the texts using `gutenberg_download()`.

```

bronte <- gutenberg_download(c(1260, 768, 969, 9182, 767))

tidy_bronte <- bronte %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)

```

What are the most common words in these novels of the Brontë sisters?

```

tidy_bronte %>%
  count(word, sort = TRUE)

## # A tibble: 23,051 × 2
##   word     n
##   <chr> <int>
## 1 time    1065
## 2 miss    855
## 3 day     827
## 4 hand    768
## 5 eyes    713
## 6 night   647
## 7 heart   638
## 8 looked   602
## 9 door    592
## 10 half    586
## # ... with 23,041 more rows

```

Interesting that "time," "eyes," and "hand" are in the top 10 for both H.G. Wells and the Brontë sisters. → Well don't stop there! CLA is literary analysis too; what does it MEAN?

Now, let's calculate the frequency for each word in the works of Jane Austen, the Brontë sisters, and H.G. Wells by binding the data frames together. We can use spread and gather from tidyverse to reshape our data frame so that it is just what we need for plotting and comparing the three sets of novels.

```

library(tidyverse)

frequency <- bind_rows(mutate(tidy_bronte, author = "Brontë Sisters"),
                        mutate(tidy_hgwells, author = "H.G. Wells"),
                        mutate(tidy_books, author = "Jane Austen")) %>%
  mutate(word = str_extract(word, "[a-z']+")) %>%
  count(author, word) %>%
  group_by(author) %>%
  mutate(proportion = n / sum(n)) %>%
  select(-n) %>%
  spread(author, proportion) %>%
  gather(author, proportion, `Brontë Sisters`:H.G. Wells) → This is on top left

```

We use `str_extract()` here because the UTF-8 encoded texts from Project Gutenberg have some examples of words with underscores around them to indicate emphasis (like italics). The tokenizer treated these as words, but we don't want to count "any" separately from "any" as we saw in our initial data exploration before choosing to use `str_extract()`.

Now let's plot (Figure 1-3).

Why are
there
missing
values?

```
library(scales)

# expect a warning about rows with missing values being removed
ggplot(frequency, aes(x = proportion, y = `Jane Austen`,
color = abs(`Jane Austen` - proportion))) +
geom_abline(color = "gray40", lty = 2) +
geom_jitter(alpha = 0.1, size = 2.5, width = 0.3, height = 0.3) +
geom_text(aes(label = word), check_overlap = TRUE, vjust = 1.5) +
scale_x_log10(labels = percent_format()) +
scale_y_log10(labels = percent_format()) +
scale_color_gradient(limits = c(0, 0.001),
low = "darkslategray4", high = "gray75") +
facet_wrap(~author, ncol = 2) +
theme(legend.position="none") +
labs(y = "Jane Austen", x = NULL)
```



Figure 1-3. Comparing the word frequencies of Jane Austen, the Brontë sisters, and H.G. Wells

Words that are close to the line in these plots have similar frequencies in both sets of texts, for example, in both Austen and Brontë texts ("miss," "time," and "day" at the high frequency end) or in both Austen and Wells texts ("time," "day," and "brother" at the high frequency end). Words that are far from the line are words that are found more in one set of texts than another. For example, in the Austen-Brontë panel, words like "elizabeth," "emma," and "fanny" (all proper nouns) are found in Austen's texts but not much in the Brontë texts, while words like "arthur" and "dog" are found in the Brontë texts but not the Austen texts. In comparing H.G. Wells with Jane Aus-

ten, Wells uses words like "beast," "guns," "feet," and "black" that Austen does not, while Austen uses words like "family," "friend," "letter," and "dear" that Wells does not.

Overall, notice in Figure 1-3 that the words in the Austen-Brontë panel are closer to the zero-slope line than in the Austen-Wells panel. Also notice that the words extend to lower frequencies in the Austen-Brontë panel; there is empty space in the Austen-Wells panel at low frequency. These characteristics indicate that Austen and the Brontë sisters use more similar words than Austen and H.G. Wells. Also, we see that not all the words are found in all three sets of texts, and there are fewer data points in the panel for Austen and H.G. Wells.

Let's quantify how similar and different these sets of word frequencies are using a correlation test. How correlated are the word frequencies between Austen and the Brontë sisters, and between Austen and Wells?

```
cor.test(data = frequency[frequency$author == "Brontë Sisters",],  
         ~ proportion + `Jane Austen`)  
  
##  
## Pearson's product-moment correlation  
##  
## data: proportion and Jane Austen  
## t = 119.64, df = 10404, p-value < 2.2e-16  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.7527837 0.7689611 ✓  
## sample estimates:  
## cor  
## 0.7609907  
  
cor.test(data = frequency[frequency$author == "H.G. Wells",],  
         ~ proportion + `Jane Austen`)  
  
##  
## Pearson's product-moment correlation  
##  
## data: proportion and Jane Austen  
## t = 36.441, df = 6053, p-value < 2.2e-16  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.4032820 0.4446006 ✓  
## sample estimates:  
## cor  
## 0.424162
```

Just as we saw in the plots, the word frequencies are more correlated between the Austen and Brontë novels than between Austen and H.G. Wells.

I got a cor.value of ≈ 0.41377 . Why ...?

CHAPTER 2

Sentiment Analysis with Tidy Data

In the previous chapter, we explored in depth what we mean by the tidy text format and showed how this format can be used to approach questions about word frequency. This allowed us to analyze which words are used most frequently in documents and to compare documents, but now let's investigate a different topic. Let's address the topic of "opinion mining" or "sentiment analysis." When human readers approach a text, we use our understanding of the emotional intent of words to infer whether a section of text is positive or negative, or perhaps characterized by some other more nuanced emotion like surprise or disgust. We can use the tools of text mining to approach the emotional content of text programmatically, as shown in Figure 2-1.

Is there a way to create my own dictionary for this, or perhaps to edit a sentiment dictionary package in R? For ex, Tolkien had "new" words which have special sentiments which (I guess) are not already in present sentiment dictionaries

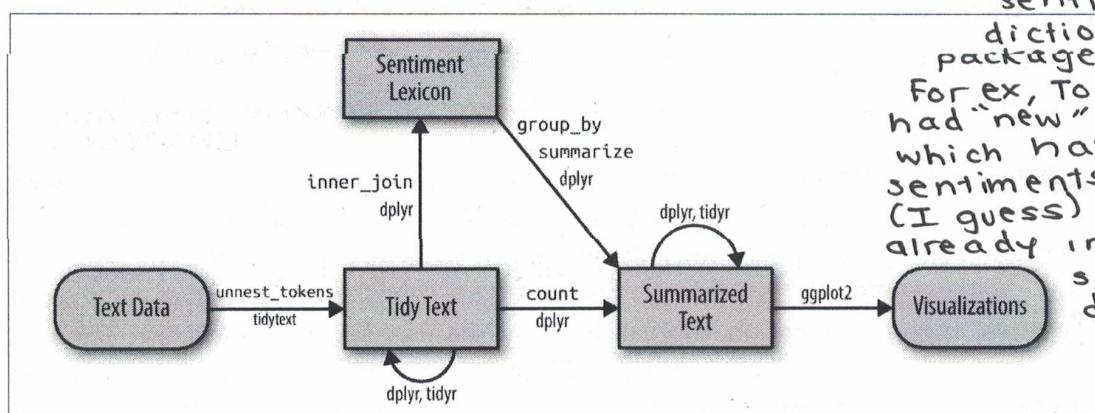


Figure 2-1. A flowchart of a typical text analysis that uses tidytext for sentiment analysis. This chapter shows how to implement sentiment analysis using tidy data principles.

One way to analyze the sentiment of a text is to consider the text as a combination of its individual words, and the sentiment content of the whole text as the sum of the sentiment content of the individual words. This isn't the only way to approach senti-

$\sum (\text{individual sent})$

ment analysis, but it is an often-used approach, *and* an approach that naturally takes advantage of the tidy tool ecosystem.

The sentiments Dataset

As discussed above, there are a variety of methods and dictionaries that exist for evaluating opinion or emotion in text. The tidytext package contains several sentiment lexicons in the sentiments dataset.

"lexicon" is the vocabulary of a specific branch of knowledge

```
library(tidytext)
```

sentiments

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

when I run the "sentiments" command, I get a different set of words. Is this bc the dataset has been updated?

The three general-purpose lexicons are:

- AFINN from Finn Årup Nielsen → -5 to +5 SCORE
- Bing from Bing Liu and collaborators → POSITIVE / NEGATIVE
- NRC from Saif Mohammad and Peter Turney → YES / NO FOR ALL EMOTIONS

All three lexicons are based on unigrams, i.e., single words. These lexicons contain many English words and the words are assigned scores for positive/negative sentiment, and also possibly emotions like joy, anger, sadness, and so forth. The NRC lexicon categorizes words in a binary fashion ("yes"/"no") into categories of positive, negative, anger, anticipation, disgust, fear, joy, sadness, surprise, and trust. The Bing lexicon categorizes words in a binary fashion into positive and negative categories. The AFINN lexicon assigns words with a score that runs between -5 and 5, with negative scores indicating negative sentiment and positive scores indicating positive sentiment. All of this information is tabulated in the sentiments dataset, and tidytext provides the function `get_sentiments()` to get specific sentiment lexicons without the columns that are not used in that lexicon.

these three general-purpose lexicons might NOT work on Tolkien CLA.

```

get_sentiments("afinn")
## # A tibble: 2,476 × 2
##       word score
##       <chr> <int>
## 1 abandon    -2
## 2 abandoned   -2
## 3 abandons   -2
## 4 abducted   -2
## 5 abduction  -2
## 6 abductions -2
## 7 abhor     -3
## 8 abhorred   -3
## 9 abhorrent  -3
## 10 abhors    -3
## # ... with 2,466 more rows ✓

get_sentiments("bing")
## # A tibble: 6,788 × 2
##       word sentiment
##       <chr>    <chr>
## 1 2-faced  negative
## 2 2-faces  negative
## 3 a+       positive
## 4 abnormal negative
## 5 abolish  negative
## 6 abominable negative
## 7 abominably negative
## 8 abominate negative
## 9 abomination negative
## 10 abort    negative
## # ... with 6,778 more rows ✓

get_sentiments("nrc")
## # A tibble: 13,901 × 2
##       word sentiment
##       <chr>    <chr>
## 1 abacus   trust
## 2 abandon   fear
## 3 abandon   negative
## 4 abandon   sadness
## 5 abandoned anger
## 6 abandoned fear
## 7 abandoned negative
## 8 abandoned sadness
## 9 abandonment anger
## 10 abandonment fear
## # ... with 13,891 more rows ✓

```

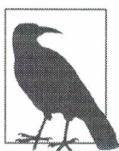
How were these sentiment lexicons put together and validated? They were constructed via either crowdsourcing (using, for example, Amazon Mechanical Turk) or by the labor of one of the authors, and were validated using some combination of

LOTS OF LIMITATIONS
↓ TO THIS SENT ANAL
BELOW

make your own lexicon
of others' input

crowdsourcing again, restaurant or movie reviews, or Twitter data. Given this information, we may hesitate to apply these sentiment lexicons to styles of text dramatically different from what they were validated on, such as narrative fiction from 200 years ago. While it is true that using these sentiment lexicons with, for example, Jane Austen's novels may give us less accurate results than with tweets sent by a contemporary writer, we still can measure the sentiment content for words that are shared across the lexicon and the text.

There are also some domain-specific sentiment lexicons available, constructed to be used with text from a specific content area. "Example: Mining Financial Articles" on page 81 explores an analysis using a sentiment lexicon specifically for finance.



Dictionary-based methods like the ones we are discussing find the total sentiment of a piece of text by adding up the individual sentiment scores for each word in the text.

Not every English word is in the lexicons because many English words are pretty neutral. It is important to keep in mind that these methods do not take into account qualifiers before a word, such as in "no good" or "not true"; a lexicon-based method like this is based on unigrams only. For many kinds of text (like the narrative examples below), there are no sustained sections of sarcasm or negated text, so this is not an important effect. Also, we can use a tidy text approach to begin to understand what kinds of negation words are important in a given text; see Chapter 9 for an extended example of such an analysis.

One last caveat is that the size of the chunk of text that we use to add up unigram sentiment scores can have an effect on an analysis. A text the size of many paragraphs can often have positive and negative sentiment averaging out to about zero, while sentence-sized or paragraph-sized text often works better.

Sentiment Analysis with Inner Join

inner join combines records from 2 tables

With data in a tidy format, sentiment analysis can be done as an inner join. This is another of the great successes of viewing text mining as a tidy data analysis task—much as removing stop words is an anti-join operation, performing sentiment analysis is an inner join operation.

(of emotion)

Let's look at the words with a joy score from the NRC lexicon. What are the most common joy words in *Emma*? First, we need to take the text of the novel and convert the text to the tidy format using `unnest_tokens()`, just as we did in "Tidying the Works of Jane Austen" on page 4. Let's also set up some other columns to keep track

whenever there's matching values in a field common to both values

of which line and chapter of the book each word comes from; we use `group_by` and `mutate` to construct those columns.

```
library(janeaustenr)
library(dplyr)
library(stringr)

tidy_books <- austen_books() %>%
  group_by(book) %>%
  mutate(linenumber = row_number(),
        chapter = cumsum(str_detect(text, regex("^chapter [\\d\\d]{1,2}")),
                           ignore_case = TRUE))) %>%
  ungroup() %>%
  unnest_tokens(word, text)
```

Notice that we chose the name `word` for the output column from `unnest_tokens()`. This is a convenient choice because the sentiment lexicons and stop-word datasets have columns named `word`; performing inner joins and anti-joins is thus easier.

Now that the text is in a tidy format with one word per row, we are ready to do the sentiment analysis. First, let's use the NRC lexicon and `filter()` for the joy words. Next, let's `filter()` the data frame with the text from the book for the words from *Emma* and then use `inner_join()` to perform the sentiment analysis. What are the most common joy words in *Emma*? Let's use `count()` from `dplyr`.

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

tidy_books %>%
  filter(book == "Emma") %>%
  inner_join(nrcjoy) %>%
  count(word, sort = TRUE)

## # A tibble: 303 x 2
##       word     n
##   <chr> <int>
## 1 good    359 ✓
## 2 young   192
## 3 friend  166
## 4 hope    143
## 5 happy   125
## 6 love    117
## 7 deal    92
## 8 found   92
## 9 present  89
## 10 kind   82
## # ... with 293 more rows
```

3 Again, I'm getting similar but not same results.
good, friend, hope, happy,
love...
why?

.. We see many positive, happy words about hope, friendship, and love here.

! Or instead we could examine how sentiment changes throughout each novel. We can do this with just a handful of lines that are mostly dplyr functions. First, we find a sentiment score for each word using the Bing lexicon and `inner_join()`.

Next, we count up how many positive and negative words there are in defined sections of each book. We define an `index` here to keep track of where we are in the narrative; this index (using integer division) counts up sections of 80 lines of text.



The `%/%` operator does integer division (`x %/% y` is equivalent to `floor(x/y)`) so the index keeps track of which 80-line section of text we are counting up negative and positive sentiment in.

Small sections of text may not have enough words in them to get a good estimate of sentiment, while really large sections can wash out narrative structure. For these books, using 80 lines works well, but this can vary depending on individual texts, how long the lines were to start with, etc. We then use `spread()` so that we have negative and positive sentiment in separate columns, and lastly calculate a net sentiment (`positive - negative`).

```
library(tidyr)

janeaustensentiment <- tidy_books %>%
  inner_join(get_sentiments("bing")) %>%
  count(book, index = linenumber %/%
    80, sentiment) %>%
  spread(sentiment, n, fill = 0) %>%
  mutate(sentiment = positive - negative)
```

Now we can plot these sentiment scores across the plot trajectory of each novel. Notice that we are plotting against the `index` on the x-axis that keeps track of narrative time in sections of text (Figure 2-2).

```
library(ggplot2)

ggplot(janeaustensentiment, aes(index, sentiment, fill = book)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~book, ncol = 2, scales = "free_x")
```

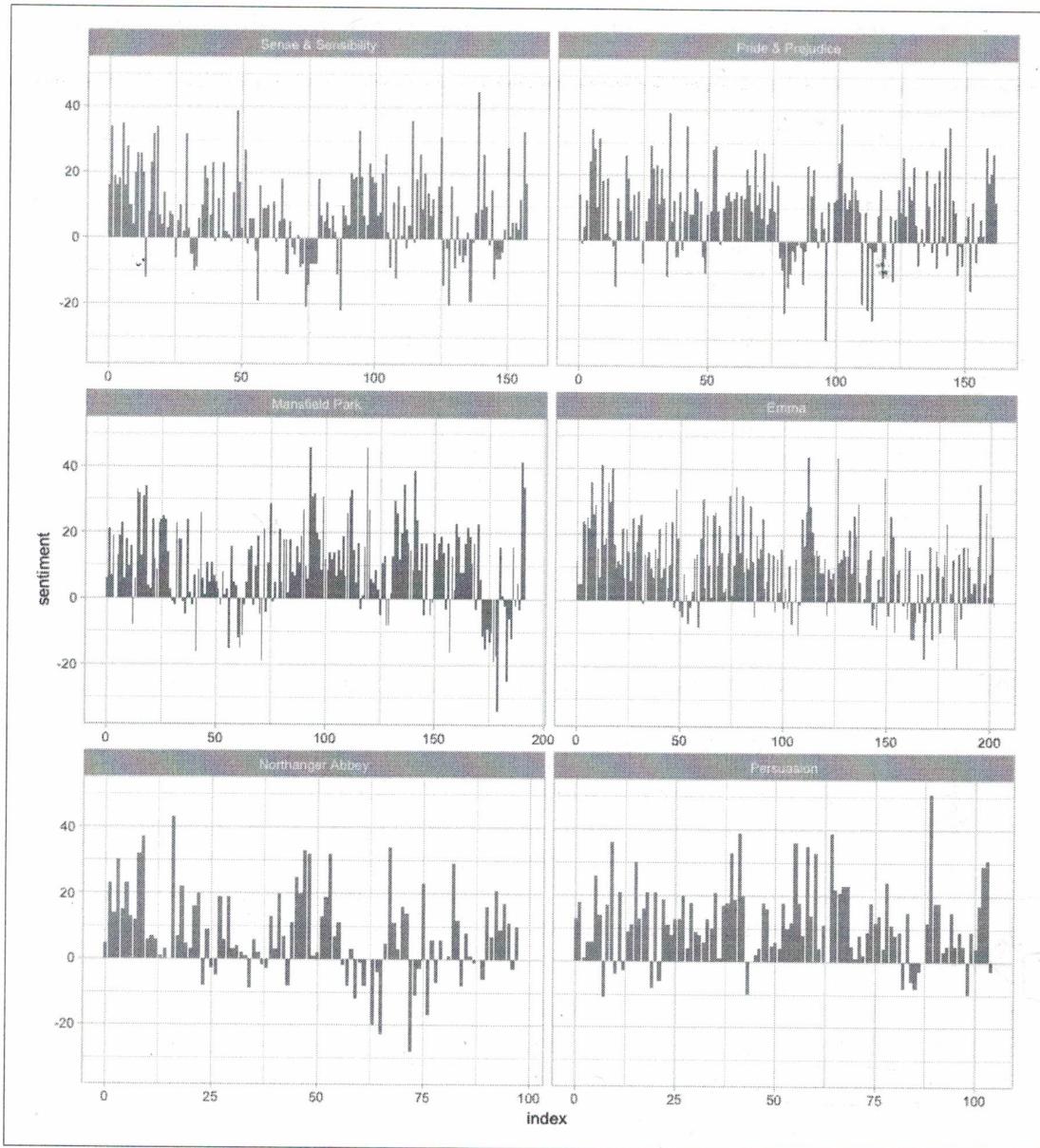


Figure 2-2. Sentiment through the narratives of Jane Austen's novels

We can see in Figure 2-2 how the plot of each novel changes toward more positive or negative sentiment over the trajectory of the story.

Comparing the Three Sentiment Dictionaries

With several options for sentiment lexicons, you might want some more information on which one is appropriate for your purposes. Let's use all three sentiment lexicons and examine how the sentiment changes across the narrative arc of *Pride and Preju-*

dice. First, let's use `filter()` to choose only the words from the one novel we are interested in.

```
pride_prejudice <- tidy_books %>%
  filter(book == "Pride & Prejudice") ✓

pride_prejudice

## # A tibble: 122,204 × 4
##   book linenumber chapter word
##   <fctr>     <int>    <int> <chr>
## 1 Pride & Prejudice      1        0 pride
## 2 Pride & Prejudice      1        0 and
## 3 Pride & Prejudice      1        0 prejudice
## 4 Pride & Prejudice      3        0 by
## 5 Pride & Prejudice      3        0 jane
## 6 Pride & Prejudice      3        0 austen
## 7 Pride & Prejudice      7        1 chapter ✓
## 8 Pride & Prejudice      7        1 1
## 9 Pride & Prejudice     10        1 it
## 10 Pride & Prejudice     10        1 is
## # ... with 122,194 more rows
```

Now, we can use `inner_join()` to calculate the sentiment in different ways.



Remember from above that the AFINN lexicon measures sentiment with a numeric score between -5 and 5, while the other two lexicons categorize words in a binary fashion, either positive or negative. To find a sentiment score in chunks of text throughout the novel, we will need to use a different pattern for the AFINN lexicon than for the other two.

vs.

Let's again use integer division (%/%) to define larger sections of text that span multiple lines, and we can use the same pattern with `count()`, `spread()`, and `mutate()` to find the net sentiment in each of these sections of text.

```
afinn <- pride_prejudice %>%
  inner_join(get_sentiments("afinn")) %>% ✓
  group_by(index = linenumber %% 80) %>%
  summarise(sentiment = sum(score)) %>% ✓
  mutate(method = "AFINN")

bing_and_nrc <- bind_rows(
  pride_prejudice %>%
    inner_join(get_sentiments("bing")) %>%
    mutate(method = "Bing et al."),
  pride_prejudice %>%
    inner_join(get_sentiments("nrc")) %>%
    filter(sentiment %in% c("positive",
                            "negative"))) %>%
    mutate(method = "NRC")) %>%
```

R had an
error here.
Line 239
of your
code.

How come
line 245
runs and says "object (score)
not found
when line 244 ran OK with (score)?

```

count(method, index = linenumber %/% 80, sentiment) %>%
spread(sentiment, n, fill = 0) %>%
mutate(sentiment = positive - negative) ✓

```

We now have an estimate of the net sentiment (positive - negative) in each chunk of the novel text for each sentiment lexicon. Let's bind them together and visualize them in Figure 2-3. ✓

```

bind_rows(afinn,
          bing_and_nrc) %>%
ggplot(aes(index, sentiment, fill = method)) +
geom_col(show.legend = FALSE) +
facet_wrap(~method, ncol = 1, scales = "free_y")

```

2
Of course here I could not graph this since I did not have "afinn" from previous error.

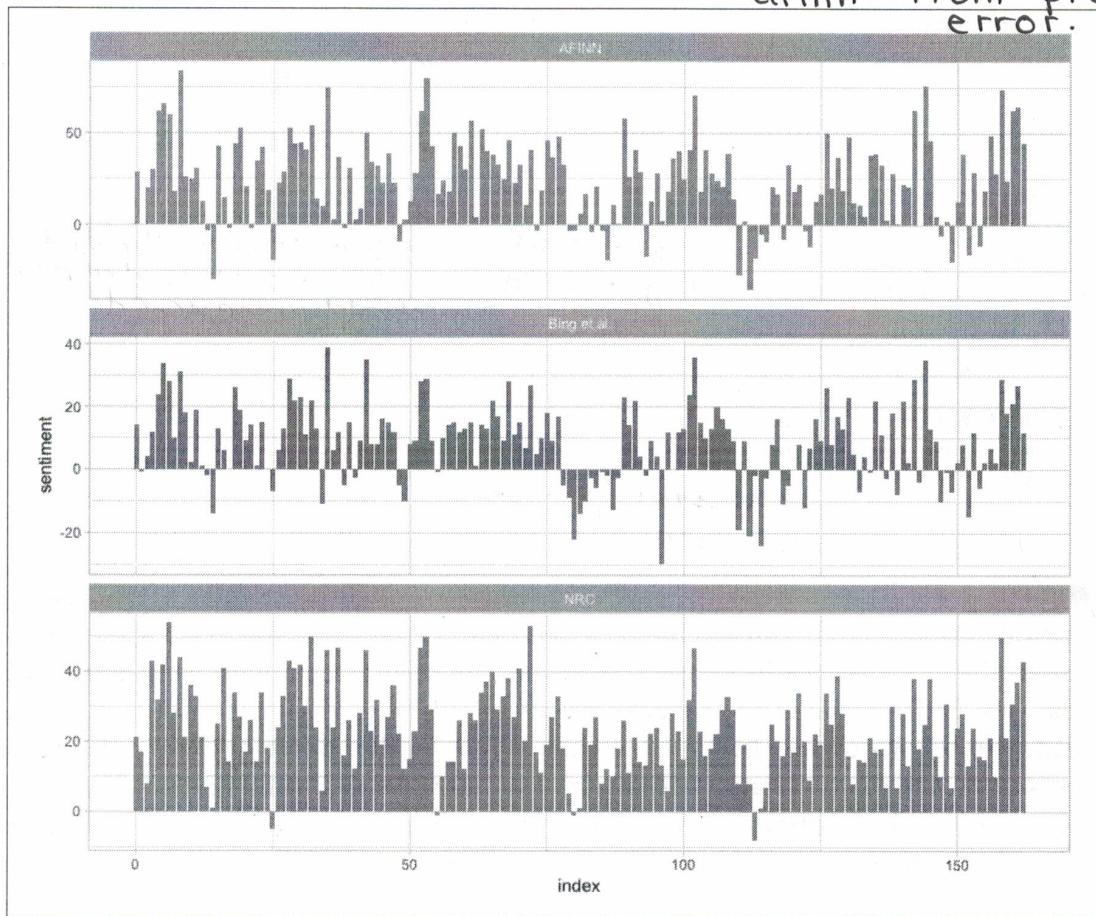


Figure 2-3. Comparing three sentiment lexicons using *Pride and Prejudice*

The three different lexicons for calculating sentiment give results that are different in an absolute sense but have similar relative trajectories through the novel. We see similar dips and peaks in sentiment at about the same places in the novel, but the absolute values are significantly different. The AFINN lexicon gives the largest absolute values, with high positive values. The lexicon from Bing et al. has lower absolute val-

ues and seems to label larger blocks of contiguous positive or negative text. The NRC results are shifted higher relative to the other two, labeling the text more positively, but detects similar relative changes in the text. We find similar differences between the methods when looking at other novels; the NRC sentiment is high, the AFINN sentiment has more variance, and the Bing et al. sentiment appears to find longer stretches of similar text, but all three agree roughly on the overall trends in the sentiment through a narrative arc.

Why is, for example, the result for the NRC lexicon biased so high in sentiment compared to the Bing et al. result? Let's look briefly at how many positive and negative words are in these lexicons.

```
get_sentiments("nrc") %>%
  filter(sentiment %in% c("positive",
                           "negative")) %>%
  count(sentiment)

## # A tibble: 2 × 2
##   sentiment     n
##   <chr>    <int>
## 1 negative  3324 ✓ 1 got 3318 ) not too drastically diff
## 2 positive  2312 ✓ 1 got 2308 so I'm OK-the dataset
                           probably changed.

get_sentiments("bing") %>%
  count(sentiment)

## # A tibble: 2 × 2
##   sentiment     n
##   <chr>    <int>
## 1 negative  4782 ✓ 4781 ) "
## 2 positive  2006 ✓ 2005
```



ie both are skewed left but Bing is more skewed

Both lexicons have more negative than positive words, but the ratio of negative to positive words is higher in the Bing lexicon than the NRC lexicon. This will contribute to the effect we see in the plot above, as will any systematic difference in word matches, for example, if the negative words in the NRC lexicon do not match very well with the words that Jane Austen uses. Whatever the source of these differences, we see similar relative trajectories across the narrative arc, with similar changes in slope, but marked differences in absolute sentiment from lexicon to lexicon. This is important context to keep in mind when choosing a sentiment lexicon for analysis.

Most Common Positive and Negative Words

One advantage of having the data frame with both sentiment and word is that we can analyze word counts that contribute to each sentiment. By implementing `count()` here with arguments of both word and sentiment, we find out how much each word contributed to each sentiment. (we can see the weight).

```

bing_word_counts <- tidy_books %>%
  inner_join(get_sentiments("bing")) %>%
  count(word, sentiment, sort = TRUE) %>%
  ungroup()

bing_word_counts

## # A tibble: 2,585 × 3
##   word    sentiment     n
##   <chr>   <chr> <int>
## 1 miss    negative  1855 ✓
## 2 well    positive  1523 ✓
## 3 good    positive  1380 ✓
## 4 great   positive  981 ✓
## 5 like    positive  725 ✓
## 6 better   positive  639
## 7 enough   positive  613
## 8 happy    positive  534
## 9 love     positive  495
## 10 pleasure positive  462
## # ... with 2,575 more rows

```

got these exact numbers ✓

This can be shown visually, and we can pipe straight into ggplot2, if we like, because of the way we are consistently using tools built for handling tidy data frames (Figure 2-4). ← this is wonderful

```

bing_word_counts %>%
  group_by(sentiment) %>%
  top_n(10) %>%
  ungroup() %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(word, n, fill = sentiment)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free_y") +
  labs(y = "Contribution to sentiment",
       x = NULL) +
  coord_flip()

```

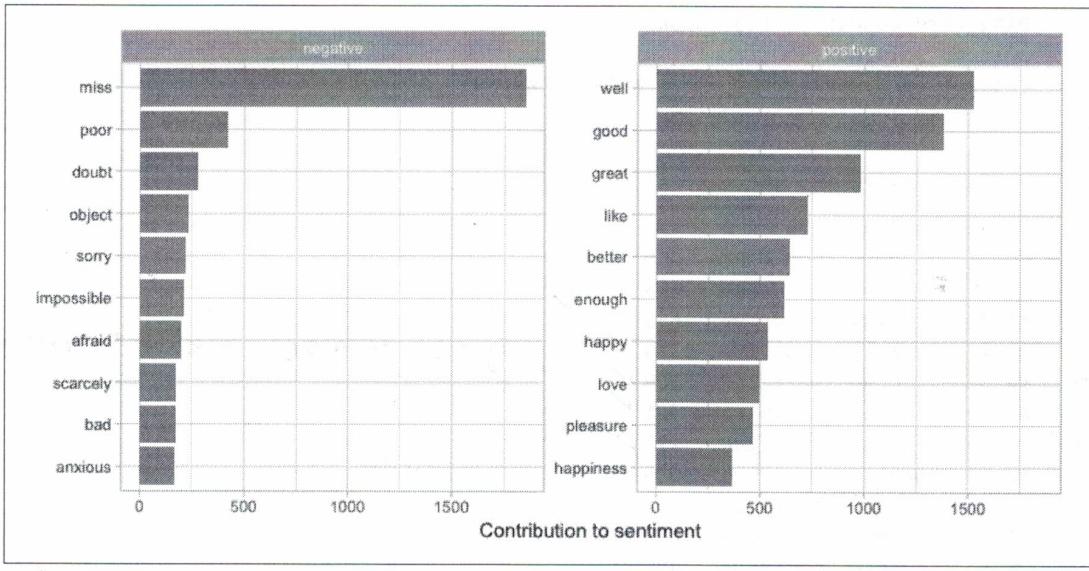


Figure 2-4. Words that contribute to positive and negative sentiment in Jane Austen's novels

Figure 2-4 lets us spot an anomaly in the sentiment analysis; the word "miss" is coded as negative but it is used as a title for young, unmarried women in Jane Austen's works. If it were appropriate for our purposes, we could easily add "miss" to a custom stop-words list using `bind_rows()`. We could implement that with a strategy such as this:

```
custom_stop_words <- bind_rows(data_frame(word = c("miss"),
                                         lexicon = c("custom")),
                                         stop_words)
```

custom_stop_words

```
## # A tibble: 1,150 × 2
##       word lexicon
##   <chr>  <chr>
## 1 miss   custom
## 2 a      SMART
## 3 a's    SMART
## 4 able   SMART
## 5 about  SMART
## 6 above  SMART
## 7 according SMART
## 8 accordingly SMART
## 9 across  SMART
## 10 actually SMART
## # ... with 1,140 more rows
```

good, so these are a list of all of the stopwords used /removed from our text when we use the command "anti-join"

Wordclouds

We've seen that this tidy text mining approach works well with ggplot2, but having our data in a tidy format is useful for other plots as well.

For example, consider the `wordcloud` package, which uses base R graphics. Let's look at the most common words in Jane Austen's works as a whole again, but this time as a wordcloud in Figure 2-5.

```
library(wordcloud)  
tidy_books %>%  
  anti_join(stop_words) %>% ✓  
  count(word) %>%  
  with(wordcloud(word, n, max.words = 100))
```

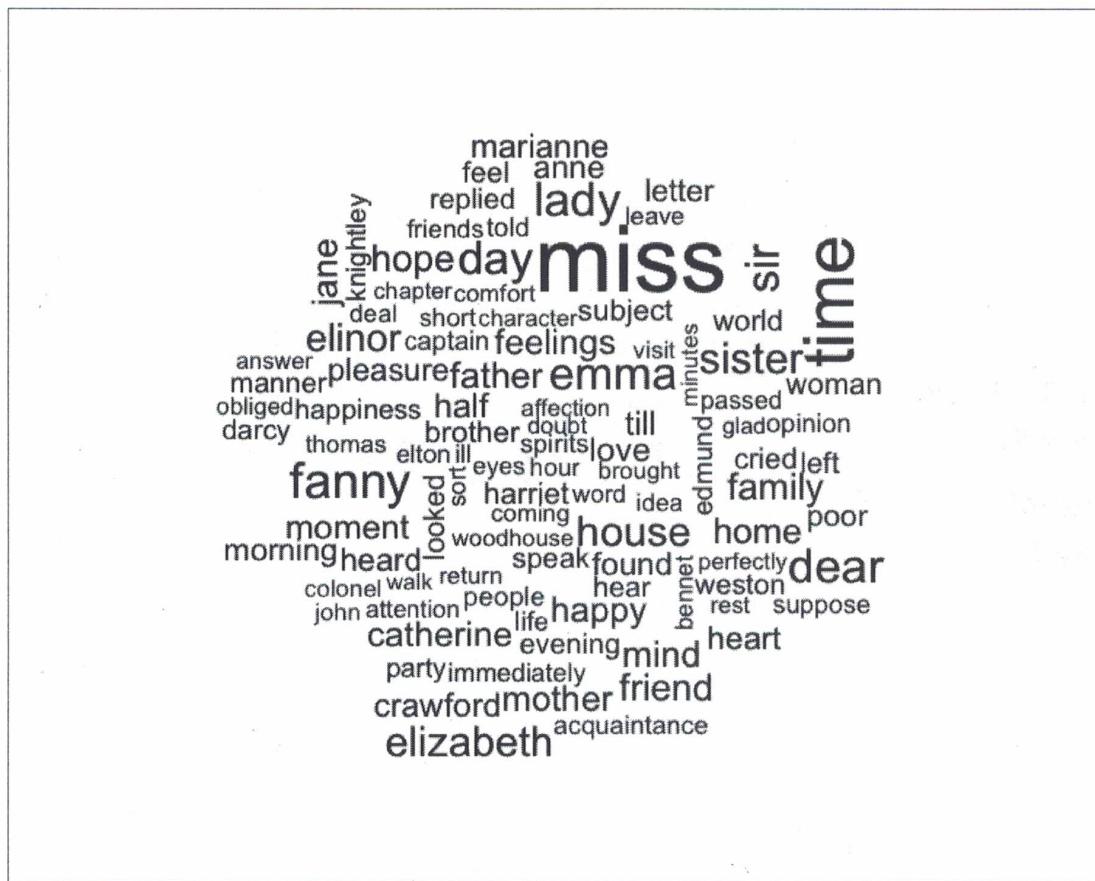


Figure 2-5. The most common words in Jane Austen's novels

In other functions, such as `comparison.cloud()`, you may need to turn the data frame into a matrix with `reshape2`'s `acast()`. Let's do the sentiment analysis to tag positive and negative words using an inner join, then find the most common positive and negative words. Until the step where we need to send the data to `compari`

`son.cloud()`, this can all be done with joins, piping, and `dplyr` because our data is in tidy format (Figure 2-6).

```
library(reshape2)

tidy_books %>%
  inner_join(get_sentiments("bing")) %>%
  count(word, sentiment, sort = TRUE) %>%
  acast(word ~ sentiment, value.var = "n", fill = 0) %>%
  comparison.cloud(colors = c("gray20", "gray80"),
                    max.words = 100)
```



Figure 2-6. Most common positive and negative words in Jane Austen's novels

The size of a word's text in Figure 2-6 is in proportion to its frequency within its sentiment. We can use this visualization to see the most important positive and negative words, but the sizes of the words are not comparable across sentiments. ✓

Looking at Units Beyond Just Words

Lots of useful work can be done by tokenizing at the word level, but sometimes it is useful or necessary to look at different units of text. For example, some sentiment analysis algorithms look beyond only unigrams (i.e., single words) to try to understand the sentiment of a sentence as a whole. These algorithms try to understand that "I am not having a good day" is a sad sentence, not a happy one, because of negation. ✓ R packages including coreNLP (Arnold and Tilton 2016), cleanNLP (Arnold 2016), and sentimentr (Rinker 2017) are examples of such sentiment analysis algorithms. For these, we may want to tokenize text into sentences, and it makes sense to use a new name for the output column in such a case.

```
PandP_sentences <- data_frame(text = prideprejudice) %>%  
  unnest_tokens(sentence, text, token = "sentences") ✓
```

Let's look at just one.

```
PandP_sentences$sentence[2]    I wrote [4] instead, I want 4th  
## [1] "however little known the feelings or views of such a man may be on his  
first entering a neighbourhood, this truth is so well fixed in the minds of  
the surrounding families, that he is considered the rightful property of some  
one or other of their daughters."
```

The sentence tokenizing does seem to have a bit of trouble with UTF-8 encoded text, especially with sections of dialogue; it does much better with punctuation in ASCII. One possibility, if this is important, is to try using `iconv()` with something like `iconv(text, to = 'latin1')` in a mutate statement before unnesting.

Another option in `unnest_tokens()` is to split into tokens using a regex pattern. We could use this, for example, to split the text of Jane Austen's novels into a data frame by chapter.

```
austen_chapters <- austen_books() %>%  
  group_by(book) %>%  
  unnest_tokens(chapter, text, token = "regex",  
    pattern = "Chapter|CHAPTER [\dIVXLC]") %>%  
  ungroup()  
  
austen_chapters %>%  
  group_by(book) %>%  
  summarise(chapters = n()) ✓  
  
## # A tibble: 6 × 2  
##       book chapters  
##   <fctr>     <int>
```

## 1 Sense & Sensibility	51
## 2 Pride & Prejudice	62 ✓ exact same #s ✓ i
## 3 Mansfield Park	49
## 4 Emma	56
## 5 Northanger Abbey	32
## 6 Persuasion	25

We have recovered the correct number of chapters in each novel (plus an “extra” row for each novel title). In the `austen_chapters` data frame, each row corresponds to one chapter.

Near the beginning of this chapter, we used a similar regex to find where all the chapters were in Austen’s novels for a tidy data frame organized by one word per row. We can use tidy text analysis to ask questions such as what are the most negative chapters in each of Jane Austen’s novels? First, let’s get the list of negative words from the Bing lexicon. Second, let’s make a data frame of how many words are in each chapter so we can normalize for chapter length. Then, let’s find the number of negative words in each chapter and divide by the total words in each chapter. For each book, which chapter has the highest proportion of negative words?

```

bingnegative <- get_sentiments("bing") %>%
  filter(sentiment == "negative")

wordcounts <- tidy_books %>%
  group_by(book, chapter) %>%
  summarize(words = n())

tidy_books %>%
  semi_join(bingnegative) %>%
  group_by(book, chapter) %>%
  summarize(negativewords = n()) %>%
  left_join(wordcounts, by = c("book", "chapter")) %>%
  mutate(ratio = negativewords/words) %>%
  filter(chapter != 0) %>%
  top_n(1) %>%
  ungroup()

## # A tibble: 6 × 5
##   book chapter negativewords words      ratio
##   <fctr>    <int>        <int> <int>      <dbl>
## 1 Sense & Sensibility     43        161  3405 0.04728341
## 2 Pride & Prejudice      34        111  2104 0.05275665
## 3 Mansfield Park          46        173  3685 0.04694708
## 4 Emma                     15        151  3340 0.04520958
## 5 Northanger Abbey         21        149  2982 0.04996647
## 6 Persuasion                 4        62  1807 0.03431101

```

These are the chapters with the most sad words in each book, normalized for number of words in the chapter. What is happening in these chapters? In Chapter 43 of *Sense and Sensibility*, Marianne is seriously ill, near death; and in Chapter 34 of *Pride and Prejudice*, Mr. Darcy proposes for the first time (so badly!). Chapter 46 of *Mansfield*

Park is almost the end, when everyone learns of Henry's scandalous adultery; Chapter 15 of *Emma* is when horrifying Mr. Elton proposes; and in Chapter 21 of *Northanger Abbey*, Catherine is deep in her Gothic faux fantasy of murder. Chapter 4 of *Persuasion* is when the reader gets the full flashback of Anne refusing Captain Wentworth, how sad she was, and what a terrible mistake she realized it to be.

Summary

Sentiment analysis provides a way to understand the attitudes and opinions expressed in texts. In this chapter, we explored how to approach sentiment analysis using tidy data principles; when text data is in a tidy data structure, sentiment analysis can be implemented as an inner join. We can use sentiment analysis to understand how a narrative arc changes throughout its course or what words with emotional and opinion content are important for a particular text. We will continue to develop our toolbox for applying sentiment analysis to different kinds of text in our case studies later in this book.



CHAPTER 3

Analyzing Word and Document Frequency: tf-idf

Term frequency,
Inverse document
frequency

ie $tf \times idf$

This can be ridiculously useful for studying Tolkiens construction of Lúthien Tinúviel (re: Clare Moore's article)

since there are multiple versions of one story, we can track how one word's importance has changed over the edits.

A central question in text mining and natural language processing is how to quantify what a document is about. Can we do this by looking at the words that make up the document? One measure of how important a word may be is its *term frequency* (*tf*), how frequently a word occurs in a document, as we examined in Chapter 1. There are words in a document, however, that occur many times but may not be important; in English, these are probably words like "the," "is," "of," and so forth. We might take the approach of adding words like these to a list of stop words and removing them before analysis, but it is possible that some of these words might be more important in some documents than others. A list of stop words is not a very sophisticated approach to adjusting term frequency for commonly used words.

Another approach is to look at a term's *inverse document frequency* (*idf*), which decreases the weight for commonly used words and increases the weight for words that are not used very much in a collection of documents. This can be combined with term frequency to calculate a term's *tf-idf* (the two quantities multiplied together), the frequency of a term adjusted for how rarely it is used. (Weighted)

so we need to think about this when developing a methodology in our research.



The statistic *tf-idf* is intended to measure how important a word is to a document in a collection (or *corpus*) of documents, for example, to one novel in a collection of novels or to one website in a collection of websites.

The statistic *tf-idf* is a rule of thumb or heuristic quantity; while it has proved useful in text mining, search engines, etc., its theoretical foundations are considered less than firm by information theory experts. The inverse document frequency for any given term is defined as:

$$idf(\text{term}) = \ln \left(\frac{n_{\text{documents}}}{n_{\text{documents containing term}}} \right)$$

We can use tidy data principles, as described in Chapter 1, to approach tf-idf analysis and use consistent, effective tools to quantify how important various terms are in a document that is part of a collection.

Term Frequency in Jane Austen's Novels

Let's start by looking at the published novels of Jane Austen and first examine term frequency, then tf-idf. We can start just by using dplyr verbs such as `group_by()` and `join()`. What are the most commonly used words in Jane Austen's novels? (Let's also calculate the total words in each novel here, for later use.)

```
library(dplyr)
library(janeaustenr)
library(tidytext)

book_words <- austen_books() %>%
  unnest_tokens(word, text) %>%
  count(book, word, sort = TRUE) %>%
  ungroup()

total_words <- book_words %>%
  group_by(book) %>%
  summarize(total = sum(n))

book_words <- left_join(book_words, total_words) → remember that this
book_words
means it will merge table 1 with any matchings in table 2
in table 2
book_words
## # A tibble: 40,379 × 4
##       book   word     n  total
##       <fctr> <chr> <int> <int>
## 1 Mansfield Park the  6206 160460 ✓
## 2 Mansfield Park to   5475 160460 ✓
## 3 Mansfield Park and  5438 160460 ✓
## 4 Emma      to   5239 160996 ✓
## 5 Emma      the  5201 160996 ✓
## 6 Emma      and  4896 160996 ✓
## 7 Mansfield Park of   4778 160460 ✓
## 8 Pride & Prejudice the 4331 122204 ✓
## 9 Emma      of   4291 160996 ✓
## 10 Pride & Prejudice to  4162 122204 ✓
## # ... with 40,369 more rows
```

exact same

* There is one row in this `book_words` data frame for each word-book combination; `n` is the number of times that word is used in that book, and `total` is the total number of words in that book. The usual suspects are here with the highest `n`, "the," "and," "to,"

ie # of times "the" occurs / total # of words in book

and so forth. In Figure 3-1, let's look at the distribution of n/total for each novel: the number of times a word appears in a novel divided by the total number of terms (words) in that novel. This is exactly what term frequency is.

```
library(ggplot2)

ggplot(book_words, aes(n/total, fill = book)) +
  geom_histogram(show.legend = FALSE) +
  xlim(NA, 0.0009) +
  facet_wrap(~book, ncol = 2, scales = "free_y")
```

Again this plot ran OK, but why is there a warning about "removed rows with missing values?"

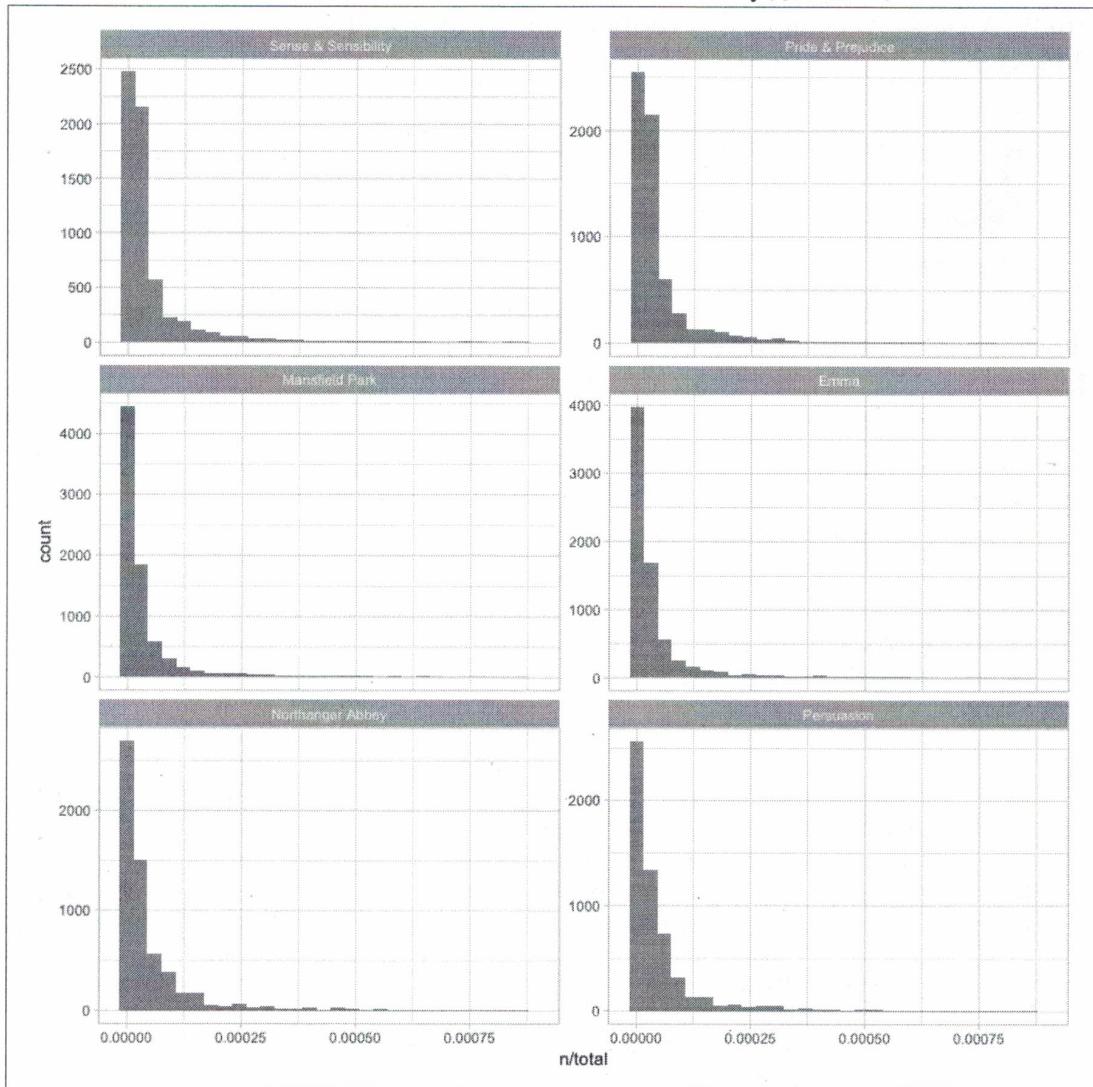


Figure 3-1. Term frequency distribution in Jane Austen's novels

There are very long tails to the right for these novels (those extremely common words!) that we have not shown in these plots. These plots exhibit similar distribu-

tions for all the novels, with many words that occur rarely and fewer words that occur frequently.

but ok what does this mean in literary analysis?

Zipf's Law

Distributions like those shown in Figure 3-1 are typical in language. In fact, those types of long-tailed distributions are so common in any given corpus of natural language (like a book, or a lot of text from a website, or spoken words) that the relationship between the frequency that a word is used and its rank has been the subject of study. A classic version of this relationship is called Zipf's law, after George Zipf, a 20th-century American linguist.



Zipf's law states that the frequency that a word appears is inversely proportional to its rank.

Since we have the data frame we used to plot term frequency, we can examine Zipf's law for Jane Austen's novels with just a few lines of dplyr functions.

```
freq_by_rank <- book_words %>%
  group_by(book) %>%
  mutate(rank = row_number(),
        `term frequency` = n/total)

freq_by_rank

## Source: local data frame [40,379 x 6]
## Groups: book [6]
##
##       book   word     n  total rank `term frequency'
##       <fctr> <chr> <int> <int> <int>      <dbl>
## 1 Mansfield Park   the  6206 160460    1  0.03867631
## 2 Mansfield Park    to  5475 160460    2  0.03412065
## 3 Mansfield Park   and  5438 160460    3  0.03389007
## 4      Emma      to  5239 160996    1  0.03254118
## 5      Emma      the  5201 160996    2  0.03230515
## 6      Emma      and  4896 160996    3  0.03041069
## 7 Mansfield Park     of  4778 160460    4  0.02977689
## 8  Pride & Prejudice   the  4331 122204    1  0.03544074
## 9      Emma      of  4291 160996    4  0.02665284
## 10  Pride & Prejudice    to  4162 122204    2  0.03405780
## # ... with 40,369 more rows
```

The rank column here tells us the rank of each word within the frequency table; the table was already ordered by n, so we could use `row_number()` to find the rank. Then, we can calculate the term frequency in the same way we did before. Zipf's law is often visualized by plotting rank on the x-axis and term frequency on the y-axis, on loga-

Same :)

V

arithmic scales. Plotting this way, an inversely proportional relationship will have a constant, negative slope (Figure 3-2).

```
freq_by_rank %>%
  ggplot(aes(rank, `term frequency`, color = book)) +
  geom_line(size = 1.1, alpha = 0.8, show.legend = FALSE) +
  scale_x_log10() +
  scale_y_log10()
```

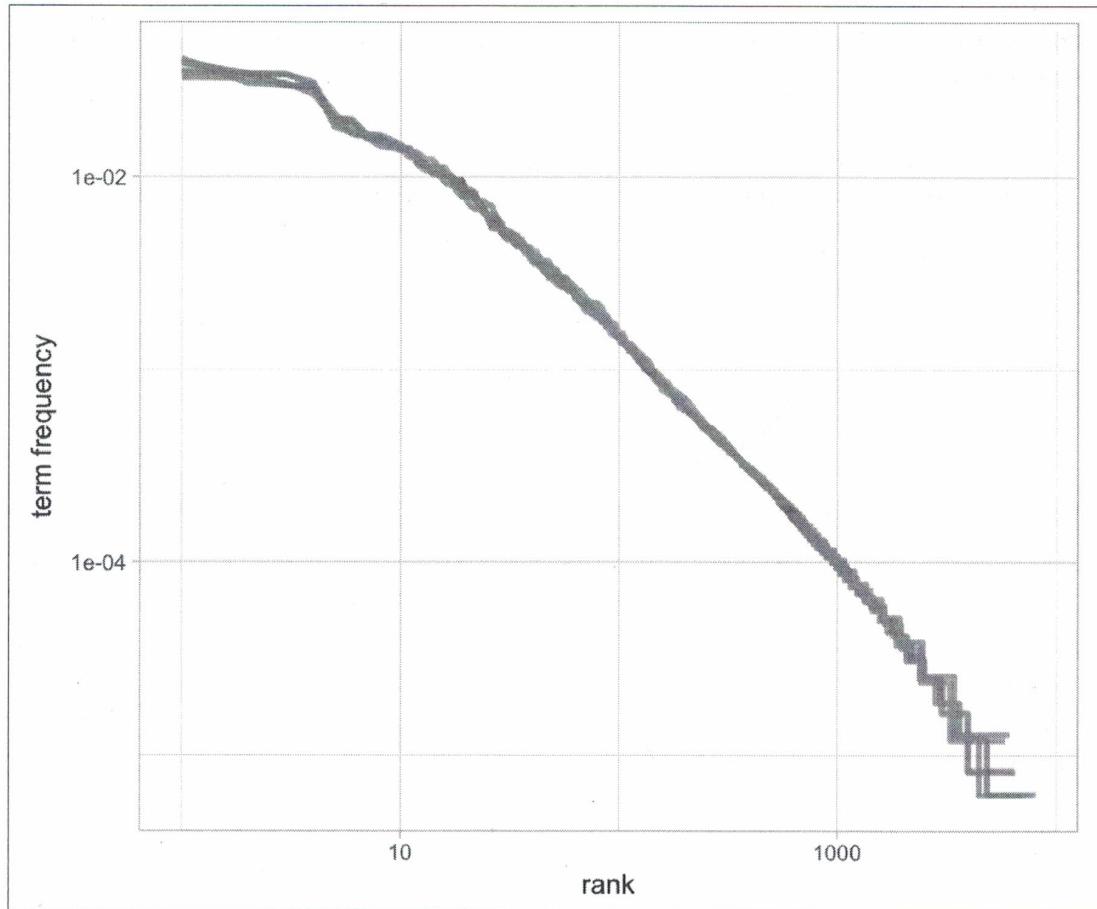


Figure 3-2. Zipf's law for Jane Austen's novels

Notice that Figure 3-2 is in log-log coordinates. We see that all six of Jane Austen's novels are similar to each other, and that the relationship between rank and frequency does have negative slope. It is not quite constant, though; perhaps we could view this as a broken power law with, say, three sections. Let's see what the exponent of the power law is for the middle section of the rank range.

```
rank_subset <- freq_by_rank %>%
  filter(rank < 500,
        rank > 10)

lm(log10(`term frequency`) ~ log10(rank), data = rank_subset)
```

gives coefficients \Rightarrow intercept = -0.6 Zipf's Law | 35
 $\log_{10}(\text{rank}) = -1.1125$

```

## 
## Call:
## lm(formula = log10(`term frequency`) ~ log10(rank), data = rank_subset)
## 
## Coefficients:
## (Intercept)  log10(rank) 
##      -0.6225     -1.1125

```

Classic versions of Zipf's law have frequency $\propto \frac{1}{\text{rank}}$ and we have in fact gotten a slope close to -1 here. Let's plot this fitted power law with the data in Figure 3-3 to see how it looks.

```

freq_by_rank %>%
  ggplot(aes(rank, `term frequency`, color = book)) +
  geom_abline(intercept = -0.62, slope = -1.1, color = "gray50", linetype = 2) +
  geom_line(size = 1.1, alpha = 0.8, show.legend = FALSE) +
  scale_x_log10() +
  scale_y_log10()

```

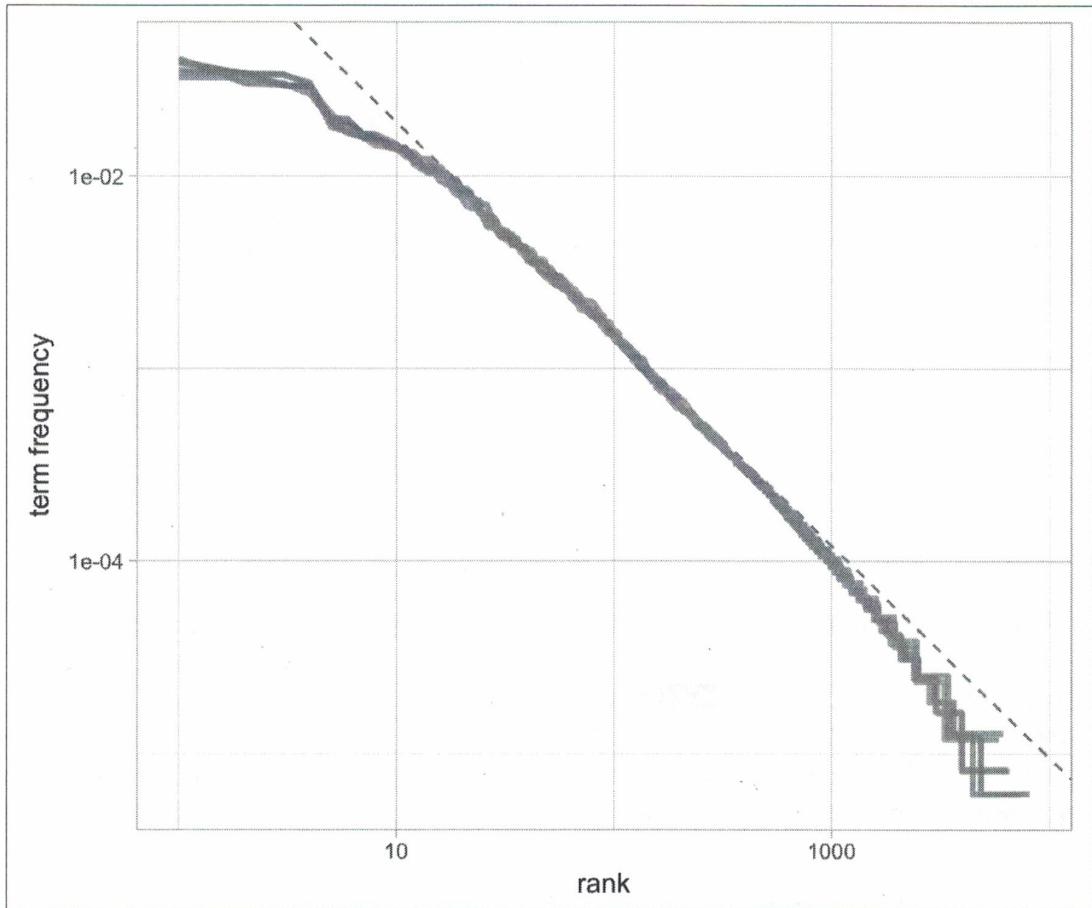


Figure 3-3. Fitting an exponent for Zipf's law with Jane Austen's novels

We have found a result close to the classic version of Zipf's law for the corpus of Jane Austen's novels. The deviations we see here at high rank are not uncommon for many kinds of language; a corpus of language often contains fewer rare words than predicted by a single power law. The deviations at low rank are more unusual. Jane Austen uses a lower percentage of the most common words than many collections of language. This kind of analysis could be extended to compare authors, or to compare any other collections of text; it can be implemented simply using tidy data principles.

The bind_tf_idf Function

The idea of tf-idf is to find the important words for the content of each document by decreasing the weight for commonly used words and increasing the weight for words that are not used very much in a collection or corpus of documents, in this case, the group of Jane Austen's novels as a whole. Calculating tf-idf attempts to find the words that are important (i.e., common) in a text, but not *too* common. Let's do that now.

The `bind_tf_idf` function in the `tidytext` package takes a tidy text dataset as input with one row per token (term), per document. One column (`word` here) contains the terms/tokens, one column contains the documents (`book` in this case), and the last necessary column contains the counts, or how many times each document contains each term (`n` in this example). We calculated a `total` for each book for our explorations in previous sections, but it is not necessary for the `bind_tf_idf` function; the table only needs to contain all the words in each document.

```
book_words <- book_words %>%
  bind_tf_idf(word, book, n)
book_words

## # A tibble: 40,379 × 7
##       book   word     n   total      tf     idf    tf_idf
##   <fctr> <chr> <int> <int>    <dbl>    <dbl>    <dbl>
## 1 Mansfield Park   the  6206 160460  0.03867631     0     0
## 2 Mansfield Park    to  5475 160460  0.03412065     0     0
## 3 Mansfield Park   and  5438 160460  0.03389007     0     0
## 4        Emma      to  5239 160996  0.03254118     0     0
## 5        Emma      the  5201 160996  0.03230515     0     0
## 6        Emma      and  4896 160996  0.03041069     0     0
## 7 Mansfield Park     of  4778 160460  0.02977689     0     0
## 8  Pride & Prejudice   the  4331 122204  0.03544074     0     0
## 9        Emma      of  4291 160996  0.02665284     0     0
## 10  Pride & Prejudice    to  4162 122204  0.03405780     0     0
## # ... with 40,369 more rows
```

Notice that `idf` and thus `tf_idf` are zero for these extremely common words. These are all words that appear in all six of Jane Austen's novels, so the `idf` term (which will then be the natural log of 1) is zero. The inverse document frequency (and thus `tf_idf`) is very low (near zero) for words that occur in many of the documents in a collection;

this is how this approach decreases the weight for common words. The inverse document frequency will be a higher number for words that occur in fewer of the documents in the collection.

Let's look at terms with high tf-idf in Jane Austen's works.

```
book_words %>%
  select(-total) %>%
  arrange(desc(tf_idf))

## # A tibble: 40,379 × 6
##   book      word     n      tf      idf      tf_idf
##   <fctr>    <chr> <int>    <dbl>    <dbl>    <dbl>
## 1 Sense & Sensibility elinor  623 0.005193528 1.791759 0.009305552
## 2 Sense & Sensibility marianne 492 0.004101470 1.791759 0.007348847
## 3 Mansfield Park   crawford 493 0.003072417 1.791759 0.005505032
## 4 Pride & Prejudice darcy   373 0.003052273 1.791759 0.005468939
## 5 Persuasion       elliot   254 0.003036207 1.791759 0.005440153
## 6 Emma             emma    786 0.004882109 1.098612 0.005363545
## 7 Northanger Abbey tilney   196 0.002519928 1.791759 0.004515105
## 8 Emma             weston   389 0.002416209 1.791759 0.004329266
## 9 Pride & Prejudice bennet   294 0.002405813 1.791759 0.004310639
## 10 Persuasion      wentworth 191 0.002283132 1.791759 0.004090824
## # ... with 40,369 more rows
```

so many names means...

→ Here we see all proper nouns, names that are in fact important in these novels. None of them occur in all of the novels, and they are important, characteristic words for each text within the corpus of Jane Austen's novels.



Some of the values for idf are the same for different terms because there are six documents in this corpus and we are seeing the numerical value for $\ln(6/1)$, $\ln(6/2)$, etc.

Let's look at a visualization for these high tf-idf words in Figure 3-4.

```
book_words %>%
  arrange(desc(tf_idf)) %>%
  mutate(word = factor(word, levels = rev(unique(word)))) %>%
  group_by(book) %>%
  top_n(15) %>%
  ungroup %>%
  ggplot(aes(word, tf_idf, fill = book)) +
  geom_col(show.legend = FALSE) +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~book, ncol = 2, scales = "free") +
  coord_flip()
```

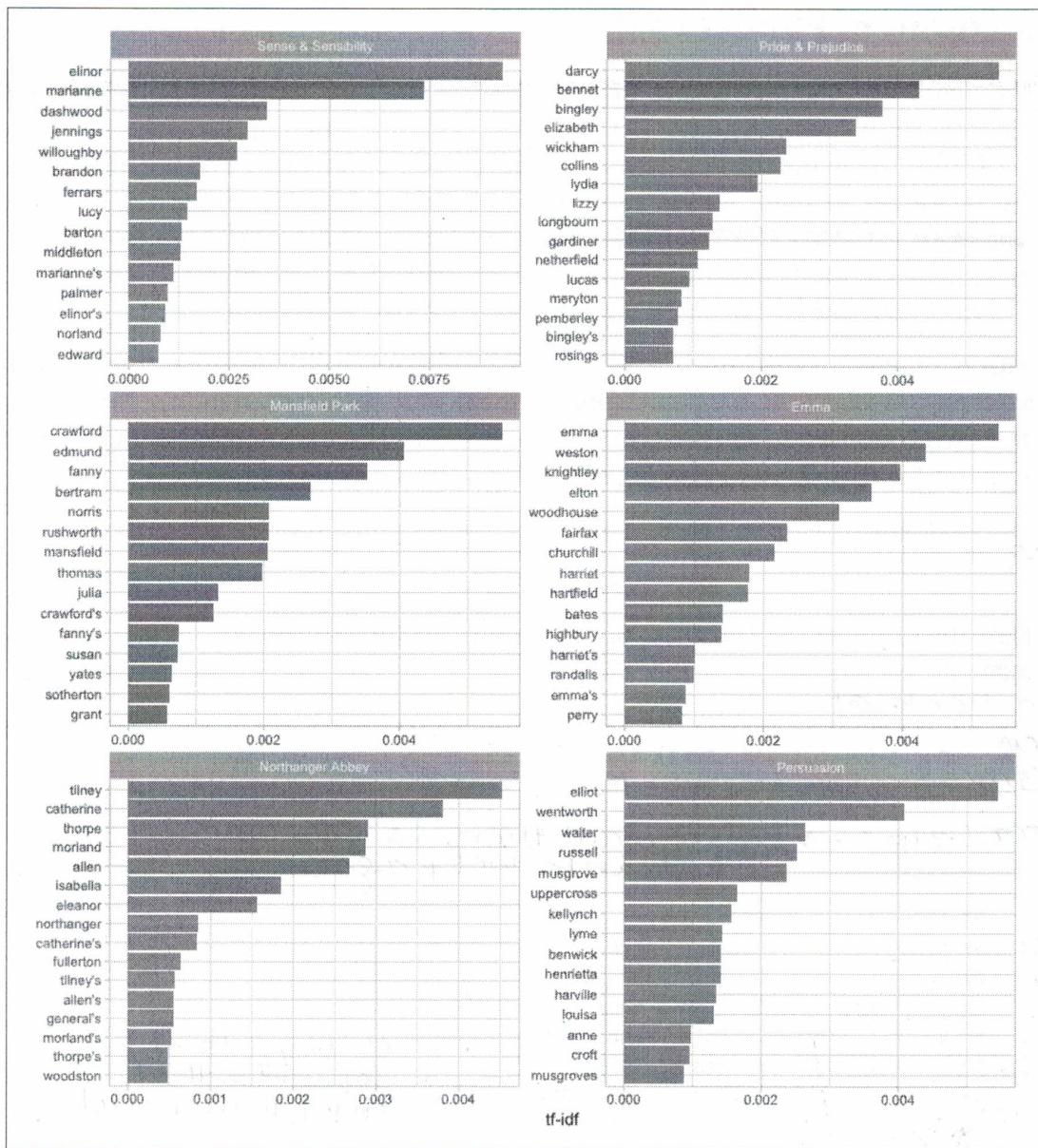


Figure 3-4. Highest tf-idf words in each of Jane Austen's novels

Still all proper nouns in Figure 3-4! These words are, as measured by tf-idf, the most important to each novel and most readers would likely agree. What measuring tf-idf has done here is show us that Jane Austen used similar language across her six novels, and what distinguishes one novel from the rest within the collection of her works are the proper nouns, the names of people and places. This is the point of tf-idf; it identifies words that are important to one document within a collection of documents.

use this
technique on
Lüthier stories!

A Corpus of Physics Texts

Let's work with another corpus of documents to see what terms are important in a different set of works. In fact, let's leave the world of fiction and narrative entirely. Let's download some classic physics texts from Project Gutenberg and see what terms are important in these works, as measured by tf-idf. Let's download *Discourse on Floating Bodies* by Galileo Galilei, *Treatise on Light* by Christiaan Huygens, *Experiments with Alternate Currents of High Potential and High Frequency* by Nikola Tesla, and *Relativity: The Special and General Theory* by Albert Einstein.

This is a pretty diverse bunch. They may all be physics classics, but they were written across a 300-year time span, and some of them were first written in other languages and then translated to English. Perfectly homogeneous these are not, but that doesn't stop this from being an interesting exercise!

Error
downloading
5001
(Einstein):

```
library(gutenbergr)
physics <- gutenberg_download(c(37729, 14725, 13476, 5001),
                               meta_fields = "author")
```

"Could not many times each word is used in each text."

download a book at
aleph.guten
berg.org/5/0/
0/5001/5001.zip" → I also went there & there was
a blank webpage?

```
## # A tibble: 12,592 × 3
##       author   word     n
##       <chr> <chr> <int>
## 1   Galilei, Galileo   the    3760
## 2      Tesla, Nikola   the    3604
## 3   Huygens, Christiaan   the    3553
## 4   Einstein, Albert   the    2994
## 5   Galilei, Galileo   of     2049
## 6   Einstein, Albert   of     2030
## 7      Tesla, Nikola   of     1737
## 8   Huygens, Christiaan   of     1708
## 9   Huygens, Christiaan   to     1207
## 10     Tesla, Nikola   a     1176
## # ... with 12,582 more rows
```

on boy, lots of
stopwords that
don't tell us much.

a i. or some need
meaningful value

Here we see just the raw counts; we need to remember that these documents are all different lengths. Let's go ahead and calculate tf-idf, then visualize the high tf-idf words in Figure 3-5.

```
plot_physics <- physics_words %>%
  bind_tf_idf(word, author, n) %>%
  arrange(desc(tf_idf)) %>%
```

```

mutate(word = factor(word, levels = rev(unique(word)))) %>%
  mutate(author = factor(author, levels = c("Galilei, Galileo",
                                             "Huygens, Christiaan",
                                             "Tesla, Nikola",
                                             "Einstein, Albert")))

plot_physics %>%
  group_by(author) %>%
  top_n(15, tf_idf) %>%
  ungroup() %>%
  mutate(word = reorder(word, tf_idf)) %>%
  ggplot(aes(word, tf_idf, fill = author)) +
  geom_col(show.legend = FALSE) +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~author, ncol = 2, scales = "free") +
  coord_flip()

```

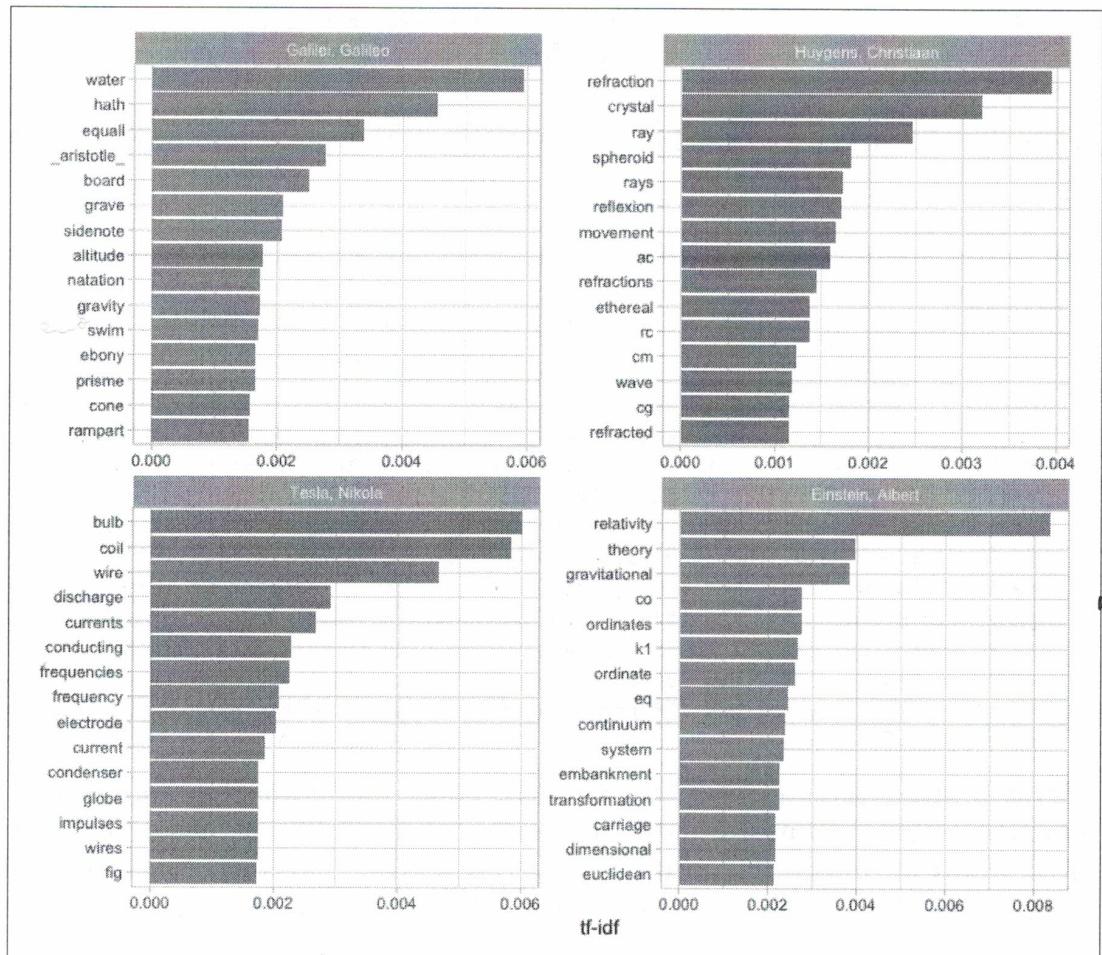


Figure 3-5. Highest tf-idf words in each physics text

Very interesting indeed. One thing we see here is “eq” in the Einstein text?! ✓

Indeed?!

```

library(stringr)

physics %>%
  filter(str_detect(text, "eq\\\\."))
  select(text)

## # A tibble: 55 × 1
#>   text
#>   <chr>
#> 1 eq. 1: file eq01.gif
#> 2 eq. 2: file eq02.gif
#> 3 eq. 3: file eq03.gif
#> 4 eq. 4: file eq04.gif
#> 5 eq. 05a: file eq05a.gif
#> 6 eq. 05b: file eq05b.gif
#> 7 the distance between the points being eq. 06 .
#> 8 direction of its length with a velocity v is eq. 06 of a metre.
#> 9 velocity v=c we should have eq. 06a ,
#> 10 the rod as judged from K1 would have been eq. 06 ;
#> # ... with 45 more rows

```

Some cleaning up of the text may be in order. “K1” is the name of a coordinate system for Einstein:

```

physics %>%
  filter(str_detect(text, "K1"))
  select(text)

## # A tibble: 59 × 1
#>   text
#>   <chr>
#> 1 to a second co-ordinate system K1 provided that the latter is
#> 2 condition of uniform motion of translation. Relative to K1 the
#> 3 tenet thus: If, relative to K, K1 is a uniformly moving co-ordinate
#> 4 with respect to K1 according to exactly the same general laws as with
#> 5 does not hold, then the Galileian co-ordinate systems K, K1, K2, etc.,
#> 6 Relative to K1, the same event would be fixed in respect of space and
#> 7 to K1, when the magnitudes x, y, z, t, of the same event with respect
#> 8 of light (and of course for every ray) with respect to K and K1. For
#> 9 reference-body K and for the reference-body K1. A light-signal is sent
#> 10 immediately follows. If referred to the system K1, the propagation of
#> # ... with 49 more rows

```

Maybe it makes sense to keep this one. Also notice that in this line we have “co-ordinate,” which explains why there are separate “co” and “ordinate” items in the high tf-idf words for the Einstein text; the `unnest_tokens()` function separates around punctuation. Notice that the tf-idf scores for “co” and “ordinate” are close to the same!

“AB,” “RC,” and so forth are names of rays, circles, angles, and so on for Huygens:

```

physics %>%
  filter(str_detect(text, "AK"))
  select(text)

```

```

## # A tibble: 34 × 1
#> #> #> text
#> #> <chr>
#> # 1 Now let us assume that the ray has come from A to C along AK, KC; the
#> # 2 be equal to the time along KMN. But the time along AK is longer than
#> # 3 that along AL: hence the time along AKN is longer than that along ABC.
#> # 4 And KC being longer than KN, the time along AKC will exceed, by as
#> # 5 line which is comprised between the perpendiculars AK, BL. Then it
#> # 6 ordinary refraction. Now it appears that AK and BL dip down toward the
#> # 7 side where the air is less easy to penetrate: for AK being longer than
#> # 8 than do AK, BL. And this suffices to show that the ray will continue
#> # 9 surface AB at the points AK_k_B. Then instead of the hemispherical
#> # 10 along AL, LB, and along AK, KB, are always represented by the line AH,
#> # ... with 24 more rows

```

Let's remove some of these less meaningful words to make a better, more meaningful plot. Notice that we make a custom list of stop words and use `anti_join()` to remove them; this is a flexible approach that can be used in many situations. We will need to go back a few steps since we are removing words from the tidy data frame (Figure 3-6).

```

mystopwords <- data_frame(word = c("eq", "co", "rc", "ac", "ak", "bn",
                                    "fig", "file", "cg", "cb", "cm"))
physics_words <- anti_join(physics_words, mystopwords, by = "word")
plot_physics <- physics_words %>%
  bind_tf_idf(word, author, n) %>%
  arrange(desc(tf_idf)) %>%
  mutate(word = factor(word, levels = rev(unique(word)))) %>%
  group_by(author) %>%
  top_n(15, tf_idf) %>%
  ungroup %>%
  mutate(author = factor(author, levels = c("Galilei, Galileo",
                                             "Huygens, Christiaan",
                                             "Tesla, Nikola",
                                             "Einstein, Albert"))))
✓

ggplot(plot_physics, aes(word, tf_idf, fill = author)) +
  geom_col(show.legend = FALSE) +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~author, ncol = 2, scales = "free") +
  coord_flip()

```

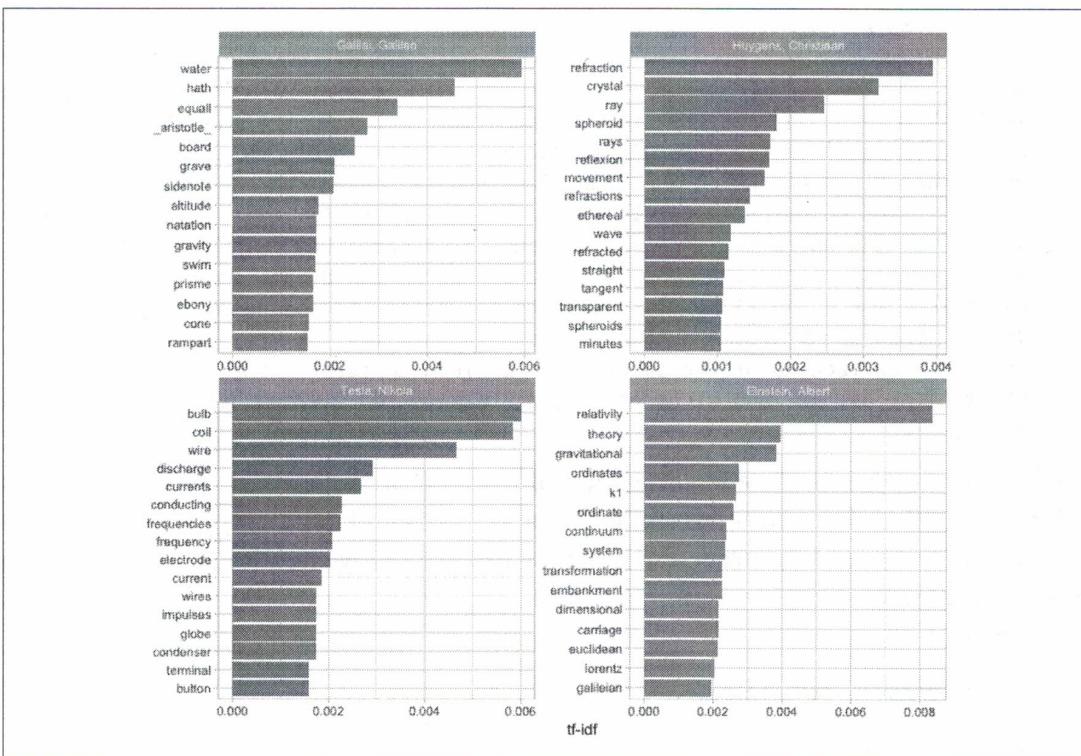


Figure 3-6. Highest tf-idf words in classic physics texts

One thing we can conclude from Figure 3-6 is that we don't hear enough about ramparts or things being ethereal in physics today. ✓

Summary

Using term frequency and inverse document frequency allows us to find words that are characteristic for one document within a collection of documents, whether that document is a novel or physics text or webpage. Exploring term frequency on its own can give us insight into how language is used in a collection of natural language, and dplyr verbs like `count()` and `rank()` give us tools to reason about term frequency. The tidytext package uses an implementation of tf-idf consistent with tidy data principles that enables us to see how different words are important in documents within a collection or corpus of documents.