My biggest worry was the removal of templates from issues. I was already  
picturing my spending hours writing regular expressions to remove these  
lines… and then I realized that the word “lines” was the key! I could go  
split all issue comments into *lines*, which is called tokenization in  
proper text mining vocabulary, and then remove duplicates! This way, I  
didn’t even have to worry about the templates having changed a bit over  
time, since each version was used at least twice. A tricky part that  
remained was the removal of code chunks since I only wanted to keep  
human conversation. In theory, it was easy: code chunks are lines  
located between two lines containing ““`”… I’m still not sure I  
solved this in the easiest possible way.

library("magrittr")

threads <- readr::read\_csv("data/clean\_data.csv")

# to remove code lines between ```

range <- function(x1, x2){

x1:x2

}

# I need the indices of lines between ```

split\_in\_indices <- function(x){

lengthx <- length(x)

if(length(x) == 0){

return(0)

}else{

if(lengthx > 2){

limits1 <- x[seq(from = 1, to = (lengthx - 1), by = 2)]

limits2 <- x[seq(from = 2, to = lengthx, by = 2)]

purrr::map2(limits1, limits2, range) %>%

unlist() %>%

c()

}else{

x[1]:x[2]

}

}

}

# tokenize by line

threads <- tidytext::unnest\_tokens(threads, line, body, token = "lines")

# remove white space

threads <- dplyr::mutate(threads, line = trimws(line))

# remove citations lines

threads <- dplyr::filter(threads, !stringr::str\_detect(line, "^\\>"))

# remove the line from the template that has ``` that used to bother me

threads <- dplyr::filter(threads, !stringr::str\_detect(line, "bounded by ```"))

# correct one line

threads <- dplyr::mutate(threads, line = stringr::str\_replace\_all(line, "`` `", "```"))

# group by comment

threads <- dplyr::group\_by(threads, title, created\_at, user, issue)

# get indices

threads <- dplyr::mutate(threads, index = 1:n())

# get lines limiting chunks

threads <- dplyr::mutate(threads, chunk\_limit = stringr::str\_detect(line, "```")&stringr::str\_count(line, "`") %in% c(3, 4))

# special treatment

threads <- dplyr::mutate(threads,

chunk\_limit = ifelse(user == "MarkEdmondson1234" & issue == 127 & index == 33,

FALSE, chunk\_limit))

threads <- dplyr::mutate(threads, which\_limit = list(split\_in\_indices(which(chunk\_limit))))

# weird code probably to get indices of code lines

threads <- dplyr::mutate(threads, code = index %in% which\_limit[[1]])

threads <- dplyr::ungroup(threads)

In the excerpt below, we see the most important variable, the binary  
code indicating whether the line is a code line. This excerpt also  
shows variables created to help compute code: index is the index of  
the line withing a comment, chunk\_limit indicates whether the line is  
a chunk limit, which\_limit indicates which indices in the comment  
indicate lines of code.

dplyr::filter(threads, package == "gutenbergr",

user == "sckott",

!stringr::str\_detect(line, "ropensci..footer")) %>%

dplyr::mutate(created\_at = as.character(created\_at)) %>%

dplyr::select(created\_at, line, code, index, chunk\_limit, which\_limit) %>%

knitr::kable()

| **created\_at** | **line** | **code** | **index** | **chunk\_limit** | **which\_limit** |
| --- | --- | --- | --- | --- | --- |
| 2016-05-02 17:04:56 | thanks for your submission @dgrtwo – seeking reviewers now | FALSE | 1 | FALSE | 0 |
| 2016-05-04 06:09:19 | reviewers: @masalmon | FALSE | 1 | FALSE | 0 |
| 2016-05-04 06:09:19 | due date: 2016-05-24 | FALSE | 2 | FALSE | 0 |
| 2016-05-12 16:16:38 | having a quick look over this now… | FALSE | 1 | FALSE | 0 |
| 2016-05-12 16:45:59 | @dgrtwo looks great. just a minor thing: | FALSE | 1 | FALSE | 3:7 |
|  |  |  |  |  |  |
| 2016-05-12 16:45:59 | “` r | TRUE | 3 | TRUE | 3:7 |
| 2016-05-12 16:45:59 | warning message: | TRUE | 4 | FALSE | 3:7 |
| 2016-05-12 16:45:59 | in node\_find\_one(x\*n**o**d\**e*, \*x\*doc, xpath = xpath, nsmap = ns) : | TRUE | 5 | FALSE | 3:7 |
| 2016-05-12 16:45:59 | 101 matches for .//a: using first | TRUE | 6 | FALSE | 3:7 |
| 2016-05-12 16:45:59 | “` | TRUE | 7 | TRUE | 3:7 |
| 2016-05-12 16:45:59 | wonder if it’s worth a suppresswarnings() there? | FALSE | 8 | FALSE | 3:7 |
| 2016-05-12 20:42:53 | great! | FALSE | 1 | FALSE | 3:5 |
| 2016-05-12 20:42:53 | – add the footer to your readme: | FALSE | 2 | FALSE | 3:5 |
| 2016-05-12 20:42:53 | “` | TRUE | 3 | TRUE | 3:5 |
| 2016-05-12 20:42:53 | “` | TRUE | 5 | TRUE | 3:5 |
| 2016-05-12 20:42:53 | – could you add url and bugreports entries to description, so people know where to get sources and report bugs/issues | FALSE | 6 | FALSE | 3:5 |
| 2016-05-12 20:42:53 | – update installation of dev versions to ropenscilabs/gutenbergr and any urls for the github repo to ropenscilabs instead of dgrtwo | FALSE | 7 | FALSE | 3:5 |
| 2016-05-12 20:42:53 | – go to the repo settings –> transfer ownership and transfer to ropenscilabs – note that all our newer pkgs go to ropenscilabs first, then when more mature we’ll move to ropensci | FALSE | 8 | FALSE | 3:5 |
| 2016-05-13 01:22:22 | nice, builds on at travis – you can keep appveyor builds under your acct, or i can start on mine, let me know | FALSE | 1 | FALSE | 0 |
| 2016-05-13 16:06:31 | updated badge link, started an appveyor account with ropenscilabs as account name – sent pr – though the build is failing, something about getting the current gutenberg url | FALSE | 1 | FALSE | 0 |
|  |  |  |  |  |  |

So as you see now getting rid of chunks is straightforward: the lines  
with code == TRUE have to be deleted.

# remove them and get rid of now useless columns

threads <- dplyr::filter(threads, !code)

threads <- dplyr::select(threads, - code, - which\_limit, - index, - chunk\_limit)

Now on to removing template parts… I noticed that removing duplicates  
was a bit too drastic because sometimes duplicates were poorly formatted  
citations in which case we definitely want to keep the first occurrence. Besides,  
duplicates were sometimes very short sentences such as “great!” that are  
not templates, that we therefore should keep. Therefore, for each line,  
I counted how many times it occurred overall (no\_total\_occ), and in  
how many issues it occurred (no\_issues).

Library(rrricanes)

dplyr::filter(threads, user == "jsta", is\_review) %>%

dplyr::select(line) %>%

head() %>%

knitr::kable()

| **line** |
| --- |
| ## package review |
| – [x] as the reviewer i confirm that there are no conflicts of interest for me to review this work (if you are unsure whether you are in conflict, please speak to your editor *before* starting your review). |
| #### documentation |
| the package includes all the following forms of documentation: |
| – [x] **a statement of need** clearly stating problems the software is designed to solve and its target audience in readme |
| – [x] **installation instructions:** for the development version of package and any non-standard dependencies in readme |

Now if we clean up a bit…

threads <- dplyr::group\_by(threads, line)

threads <- dplyr::mutate(threads, no\_total\_occ = n(),

no\_issues = length(unique(issue)),

size = stringr::str\_length(line))

threads <- dplyr::ungroup(threads)

threads <- dplyr::group\_by(threads, issue, line)

threads <- dplyr::arrange(threads, created\_at)

threads <- dplyr::filter(threads, no\_total\_occ <= 2,

# for repetitions in total keep the short ones

# bc they are stuff like "thanks" so not template

# yes 10 is arbitrary

no\_issues <= 1 | size < 10)

# when there's a duplicate in one issue

# it's probably citation

# so keep the first occurrence

get\_first <- function(x){

x[1]

}

threads <- dplyr::group\_by(threads, issue, line)

threads <- dplyr::summarise\_all(threads, get\_first)

threads <- dplyr::select(threads, - no\_total\_occ, - size, - no\_issues)

threads <- dplyr::mutate(threads, # let code words now be real words

line = stringr::str\_replace\_all(line, "`", ""),

# only keep text from links, not the links themselves

line = stringr::str\_replace\_all(line, "\\]\\(.\*\\)", ""),

line = stringr::str\_replace\_all(line, "\\[", ""),

line = stringr::str\_replace\_all(line, "blob\\/master", ""),

# ’

line = stringr::str\_replace\_all(line, "’", ""),

# remove some other links

line = stringr::str\_replace\_all(line, "https\\:\\/\\/github\\.com\\/", ""))

threads <- dplyr::filter(threads, !stringr::str\_detect(line, "estimated hours spent reviewing"))

threads <- dplyr::filter(threads, !stringr::str\_detect(line, "notifications@github\\.com"))

threads <- dplyr::filter(threads, !stringr::str\_detect(line, "reply to this email directly, view it on"))

threads <- dplyr::ungroup(threads)

Here is what we get from the same review.

dplyr::filter(threads, user == "jsta", is\_review) %>%

dplyr::select(line) %>%

head() %>%

knitr::kable()

| **line** |
| --- |
| \* also, you might consider using the skip\_on\_cran function for lines that call an external download as recommended by the ropensci packaging guide. |
| \* i am having timeout issues with building the getting\_started vignette. i wonder if there is a particular year with very few hurricanes that would solve the timeout problem. |
| \* i cannot build the data.world vignette. probably because i don’t have an api key set up. you may want to consider setting the code chunks to eval=false. |
| \* i really like the tidy\_ functions. i wonder if it would make it easier on the end-user to have the get\_ functions return tidy results by default with an optional argument to return “messy” results. |
| \* missing a maintainer field in the description |
| \* there are no examples for knots\_to\_mph, mb\_to\_in, status\_abbr\_to\_str, get\_discus, get\_fstadv, tidy\_fstadv, tidy\_wr, tidy\_fcst. maybe some can be changed to non-exported? |

So now, we mostly got the interesting human and original language.

**Computing sentiment**

Another data preparation part was to compute the sentiment score of each  
line via the sentimentr  
package by Tyler Rinker, which computes a score for sentences, not for  
single words.

Library(sentiment)

sentiment <- all %>%

dplyr::group\_by(line, created\_at, user, role, issue) %>%

dplyr::mutate(sentiment = median(sentimentr::sentiment(line)$sentiment)) %>%

dplyr::ungroup() %>%

dplyr::select(line, created\_at, user, role, issue, sentiment)

This dataset of sentiment will be used later in the post.

**Tidy text analysis of onboarding social weather**

**What do reviews talk about?**

To find out what reviews deal with as if I didn’t know about our  
guidelines, I’ll compute the frequency of words and bigrams, and the  
pairwise correlation of words within issue comments.

.

library("ggplot2")

library("ggalt")

library("hrbrthemes")

stopwords <- rcorpora::corpora("words/stopwords/en")$stopWords

word\_counts <- threads %>%

tidytext::unnest\_tokens(word, line) %>%

dplyr::filter(!word %in% stopwords) %>%

dplyr::count(word, sort = TRUE) %>%

dplyr::mutate(word = reorder(word, n))

ggplot(word\_counts[1:15,]) +

geom\_lollipop(aes(word, n),

size = 2, col = "salmon") +

hrbrthemes::theme\_ipsum(base\_size = 16,

axis\_title\_size = 16) +

coord\_flip()

Most common words in onboarding review
threads

bigrams\_counts <- threads %>%

tidytext::unnest\_tokens(bigram, line, token = "ngrams", n = 2) %>%

tidyr::separate(bigram, c("word1", "word2"), sep = " ",

remove = FALSE) %>%

dplyr::filter(!word1 %in% stopwords) %>%

dplyr::filter(!word2 %in% stopwords) %>%

dplyr::count(bigram, sort = TRUE) %>%

dplyr::mutate(bigram = reorder(bigram, n))

ggplot(bigrams\_counts[2:15,]) +

geom\_lollipop(aes(bigram, n),

size = 2, col = "salmon") +

hrbrthemes::theme\_ipsum(base\_size = 16,

axis\_title\_size = 16) +

coord\_flip()

Most common bigrams in onboarding review
threads

I’m not showing the first bigram that basically shows I’ve an encoding  
issue to solve with a variation of “´”. In any case, both figures show  
what we care about, like our guidelines that are mentioned often, and  
documentation. I think words absent from the figures such as performance  
or speed also highlight what we care less about

Code Chunks – Relationships between words : n-grams & correlations

So far we’ve considered words as individual units, and considered their relationships to sentiments or to documents. However, many interesting text analyses are based on the relationships between words, whether examining which words tend to follow others immediately, or that tend to co-occur within the same documents.

In this chapter, we’ll explore some of the methods tidytext offers for calculating and visualizing relationships between words in your text dataset. This includes the token = "ngrams" argument, which tokenizes by pairs of adjacent words rather than by individual ones. We’ll also introduce two new packages: ggraph, which extends ggplot2 to construct network plots, and widyr, which calculates pairwise correlations and distances within a tidy data frame. Together these expand our toolbox for exploring text within the tidy data framework.

## 4.1 Tokenizing by n-gram

We’ve been using the unnest\_tokens function to tokenize by word, or sometimes by sentence, which is useful for the kinds of sentiment and frequency analyses we’ve been doing so far. But we can also use the function to tokenize into consecutive sequences of words, called **n-grams**. By seeing how often word X is followed by word Y, we can then build a model of the relationships between them.

We do this by adding the token = "ngrams" option to unnest\_tokens(), and setting n to the number of words we wish to capture in each n-gram. When we set n to 2, we are examining pairs of two consecutive words, often called “bigrams”:

library(dplyr)

library(tidytext)

library(janeaustenr)

austen\_bigrams <- austen\_books() %>%

unnest\_tokens(bigram, text, token = "ngrams", n = 2)

austen\_bigrams

#> # A tibble: 675,025 × 2

#> book bigram

#> <fct> <chr>

#> 1 Sense & Sensibility sense and

#> 2 Sense & Sensibility and sensibility

#> 3 Sense & Sensibility <NA>

#> 4 Sense & Sensibility by jane

#> 5 Sense & Sensibility jane austen

#> 6 Sense & Sensibility <NA>

#> 7 Sense & Sensibility <NA>

#> 8 Sense & Sensibility <NA>

#> 9 Sense & Sensibility <NA>

#> 10 Sense & Sensibility <NA>

#> # … with 675,015 more rows

This data structure is still a variation of the tidy text format. It is structured as one-token-per-row (with extra metadata, such as book, still preserved), but each token now represents a bigram.

Notice that these bigrams overlap: “sense and” is one token, while “and sensibility” is another.

### 4.1.1 Counting and filtering n-grams

Our usual tidy tools apply equally well to n-gram analysis. We can examine the most common bigrams using dplyr’s count():

austen\_bigrams %>%

count(bigram, sort = TRUE)

#> # A tibble: 193,210 × 2

#> bigram n

#> <chr> <int>

#> 1 <NA> 12242

#> 2 of the 2853

#> 3 to be 2670

#> 4 in the 2221

#> 5 it was 1691

#> 6 i am 1485

#> 7 she had 1405

#> 8 of her 1363

#> 9 to the 1315

#> 10 she was 1309

#> # … with 193,200 more rows

As one might expect, a lot of the most common bigrams are pairs of common (uninteresting) words, such as of the and to be: what we call “stop-words” (see Chapter 1). This is a useful time to use tidyr’s separate(), which splits a column into multiple based on a delimiter. This lets us separate it into two columns, “word1” and “word2”, at which point we can remove cases where either is a stop-word.

library(tidyr)

bigrams\_separated <- austen\_bigrams %>%

separate(bigram, c("word1", "word2"), sep = " ")

bigrams\_filtered <- bigrams\_separated %>%

filter(!word1 %in% stop\_words$word) %>%

filter(!word2 %in% stop\_words$word)

# new bigram counts:

bigram\_counts <- bigrams\_filtered %>%

count(word1, word2, sort = TRUE)

bigram\_counts

#> # A tibble: 28,975 × 3

#> word1 word2 n

#> <chr> <chr> <int>

#> 1 <NA> <NA> 12242

#> 2 sir thomas 266

#> 3 miss crawford 196

#> 4 captain wentworth 143

#> 5 miss woodhouse 143

#> 6 frank churchill 114

#> 7 lady russell 110

#> 8 sir walter 108

#> 9 lady bertram 101

#> 10 miss fairfax 98

#> # … with 28,965 more rows

We can see that names (whether first and last or with a salutation) are the most common pairs in Jane Austen books.

In other analyses, we may want to work with the recombined words. tidyr’s unite() function is the inverse of separate(), and lets us recombine the columns into one. Thus, “separate/filter/count/unite” let us find the most common bigrams not containing stop-words.

bigrams\_united <- bigrams\_filtered %>%

unite(bigram, word1, word2, sep = " ")

bigrams\_united

#> # A tibble: 51,155 × 2

#> book bigram

#> <fct> <chr>

#> 1 Sense & Sensibility NA NA

#> 2 Sense & Sensibility jane austen

#> 3 Sense & Sensibility NA NA

#> 4 Sense & Sensibility NA NA

#> 5 Sense & Sensibility NA NA

#> 6 Sense & Sensibility NA NA

#> 7 Sense & Sensibility NA NA

#> 8 Sense & Sensibility NA NA

#> 9 Sense & Sensibility chapter 1

#> 10 Sense & Sensibility NA NA

#> # … with 51,145 more rows

In other analyses you may be interested in the most common trigrams, which are consecutive sequences of 3 words. We can find this by setting n = 3:

austen\_books() %>%

unnest\_tokens(trigram, text, token = "ngrams", n = 3) %>%

separate(trigram, c("word1", "word2", "word3"), sep = " ") %>%

filter(!word1 %in% stop\_words$word,

!word2 %in% stop\_words$word,

!word3 %in% stop\_words$word) %>%

count(word1, word2, word3, sort = TRUE)

#> # A tibble: 6,141 × 4

#> word1 word2 word3 n

#> <chr> <chr> <chr> <int>

#> 1 <NA> <NA> <NA> 13260

#> 2 dear miss woodhouse 20

#> 3 miss de bourgh 17

#> 4 lady catherine de 11

#> 5 poor miss taylor 11

#> 6 sir walter elliot 10

#> 7 catherine de bourgh 9

#> 8 dear sir thomas 8

#> 9 replied miss crawford 7

#> 10 sir william lucas 7

#> # … with 6,131 more rows

### 4.1.2 Analyzing bigrams

This one-bigram-per-row format is helpful for exploratory analyses of the text. As a simple example, we might be interested in the most common “streets” mentioned in each book:

bigrams\_filtered %>%

filter(word2 == "street") %>%

count(book, word1, sort = TRUE)

#> # A tibble: 33 × 3

#> book word1 n

#> <fct> <chr> <int>

#> 1 Sense & Sensibility harley 16

#> 2 Sense & Sensibility berkeley 15

#> 3 Northanger Abbey milsom 10

#> 4 Northanger Abbey pulteney 10

#> 5 Mansfield Park wimpole 9

#> 6 Pride & Prejudice gracechurch 8

#> 7 Persuasion milsom 5

#> 8 Sense & Sensibility bond 4

#> 9 Sense & Sensibility conduit 4

#> 10 Persuasion rivers 4

#> # … with 23 more rows

A bigram can also be treated as a term in a document in the same way that we treated individual words. For example, we can look at the tf-idf (Chapter 3) of bigrams across Austen novels. These tf-idf values can be visualized within each book, just as we did for words (Figure 4.1).

bigram\_tf\_idf <- bigrams\_united %>%

count(book, bigram) %>%

bind\_tf\_idf(bigram, book, n) %>%

arrange(desc(tf\_idf))

bigram\_tf\_idf

#> # A tibble: 31,397 × 6

#> book bigram n tf idf tf\_idf

#> <fct> <chr> <int> <dbl> <dbl> <dbl>

#> 1 Mansfield Park sir thomas 266 0.0244 1.79 0.0438

#> 2 Persuasion captain wentworth 143 0.0232 1.79 0.0416

#> 3 Mansfield Park miss crawford 196 0.0180 1.79 0.0322

#> 4 Persuasion lady russell 110 0.0179 1.79 0.0320

#> 5 Persuasion sir walter 108 0.0175 1.79 0.0314

#> 6 Emma miss woodhouse 143 0.0129 1.79 0.0231

#> 7 Northanger Abbey miss tilney 74 0.0128 1.79 0.0229

#> 8 Sense & Sensibility colonel brandon 96 0.0115 1.79 0.0205

#> 9 Sense & Sensibility sir john 94 0.0112 1.79 0.0201

#> 10 Emma frank churchill 114 0.0103 1.79 0.0184

#> # … with 31,387 more rows

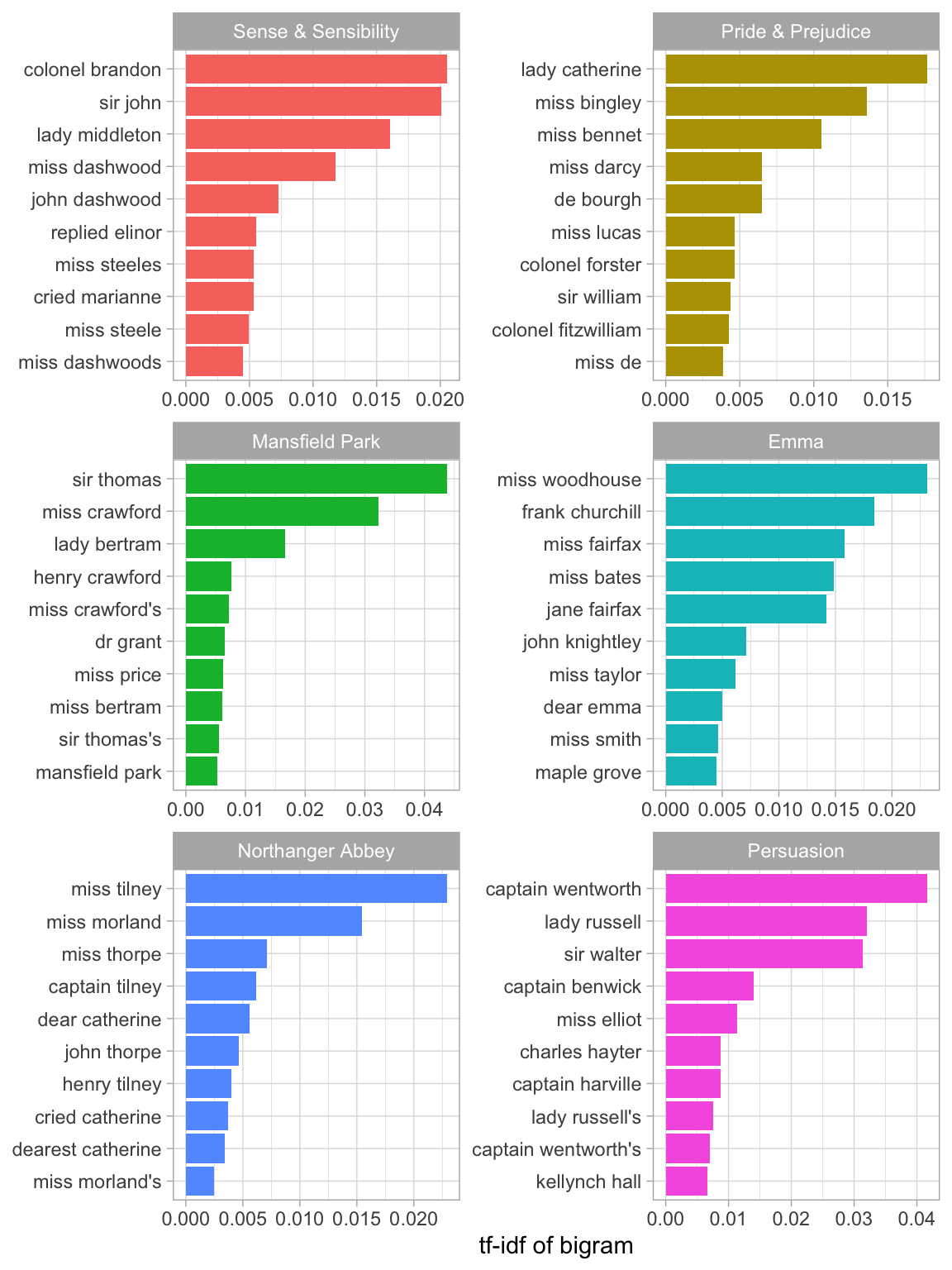


Figure 4.1: Bigrams with the highest tf-idf from each Jane Austen novel

Much as we discovered in Chapter 3, the units that distinguish each Austen book are almost exclusively names. We also notice some pairings of a common verb and a name, such as “replied elizabeth” in Pride & Prejudice, or “cried emma” in Emma.

There are advantages and disadvantages to examining the tf-idf of bigrams rather than individual words. Pairs of consecutive words might capture structure that isn’t present when one is just counting single words, and may provide context that makes tokens more understandable (for example, “pulteney street”, in Northanger Abbey, is more informative than “pulteney”). However, the per-bigram counts are also sparser: a typical two-word pair is rarer than either of its component words. Thus, bigrams can be especially useful when you have a very large text dataset.

### 4.1.3 Using bigrams to provide context in sentiment analysis

Our sentiment analysis approach in Chapter 2 simply counted the appearance of positive or negative words, according to a reference lexicon. One of the problems with this approach is that a word’s context can matter nearly as much as its presence. For example, the words “happy” and “like” will be counted as positive, even in a sentence like “I’m not **happy** and I don’t **like** it!”

Now that we have the data organized into bigrams, it’s easy to tell how often words are preceded by a word like “not”:

bigrams\_separated %>%

filter(word1 == "not") %>%

count(word1, word2, sort = TRUE)

#> # A tibble: 1,178 × 3

#> word1 word2 n

#> <chr> <chr> <int>

#> 1 not be 580

#> 2 not to 335

#> 3 not have 307

#> 4 not know 237

#> 5 not a 184

#> 6 not think 162

#> 7 not been 151

#> 8 not the 135

#> 9 not at 126

#> 10 not in 110

#> # … with 1,168 more rows

By performing sentiment analysis on the bigram data, we can examine how often sentiment-associated words are preceded by “not” or other negating words. We could use this to ignore or even reverse their contribution to the sentiment score.

Let’s use the AFINN lexicon for sentiment analysis, which you may recall gives a numeric sentiment value for each word, with positive or negative numbers indicating the direction of the sentiment.

AFINN <- get\_sentiments("afinn")

AFINN

#> # A tibble: 2,477 × 2

#> word value

#> <chr> <dbl>

#> 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 2,467 more rows

We can then examine the most frequent words that were preceded by “not” and were associated with a sentiment.

not\_words <- bigrams\_separated %>%

filter(word1 == "not") %>%

inner\_join(AFINN, by = c(word2 = "word")) %>%

count(word2, value, sort = TRUE)

not\_words

#> # A tibble: 229 × 3

#> word2 value n

#> <chr> <dbl> <int>

#> 1 like 2 95

#> 2 help 2 77

#> 3 want 1 41

#> 4 wish 1 39

#> 5 allow 1 30

#> 6 care 2 21

#> 7 sorry -1 20

#> 8 leave -1 17

#> 9 pretend -1 17

#> 10 worth 2 17

#> # … with 219 more rows

For example, the most common sentiment-associated word to follow “not” was “like”, which would normally have a (positive) score of 2.

It’s worth asking which words contributed the most in the “wrong” direction. To compute that, we can multiply their value by the number of times they appear (so that a word with a value of +3 occurring 10 times has as much impact as a word with a sentiment value of +1 occurring 30 times). We visualize the result with a bar plot (Figure 4.2).

library(ggplot2)

not\_words %>%

mutate(contribution = n \* value) %>%

arrange(desc(abs(contribution))) %>%

head(20) %>%

mutate(word2 = reorder(word2, contribution)) %>%

ggplot(aes(n \* value, word2, fill = n \* value > 0)) +

geom\_col(show.legend = FALSE) +

labs(x = "Sentiment value \* number of occurrences",

y = "Words preceded by \"not\"")

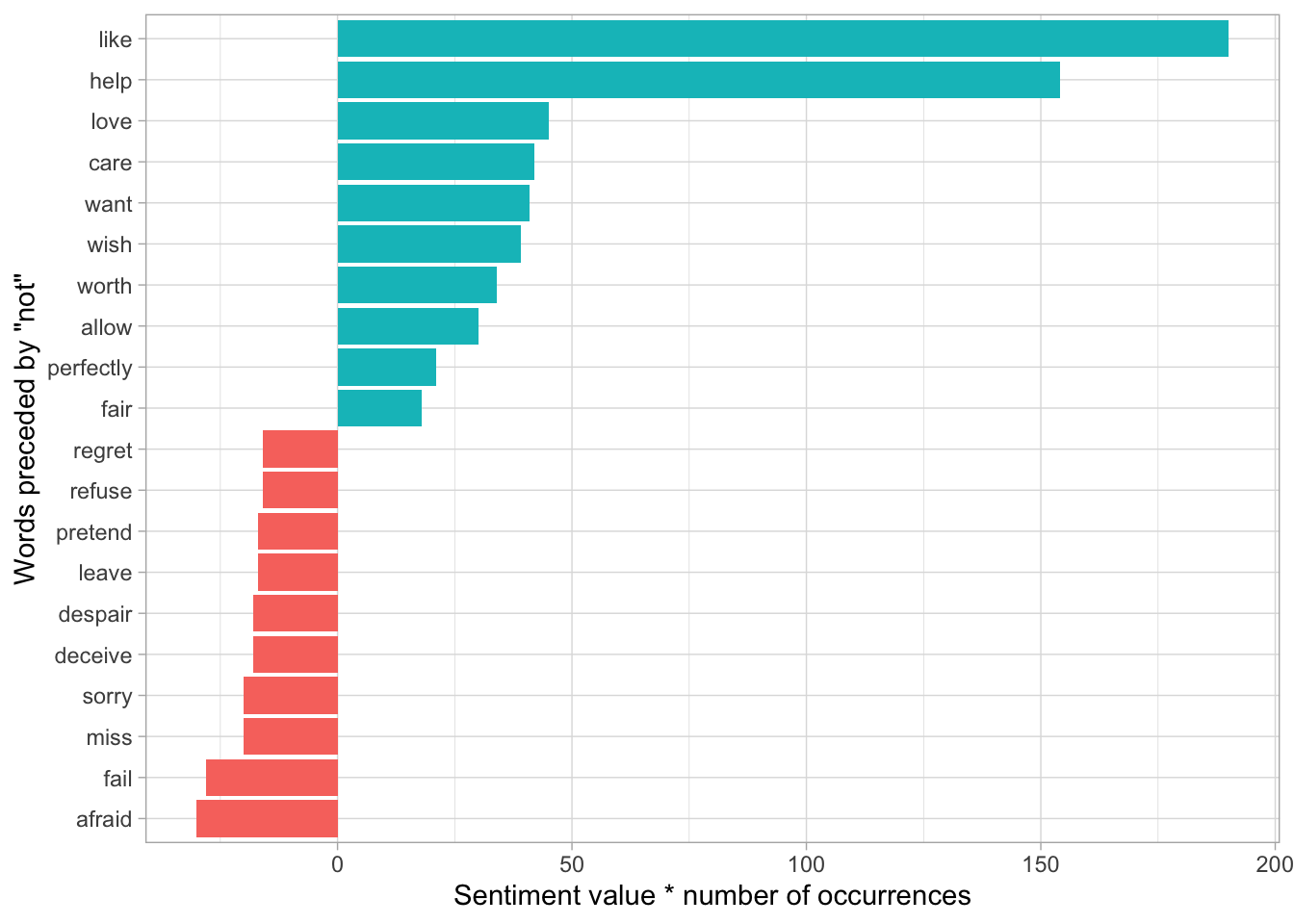


Figure 4.2: Words preceded by ‘not’ that had the greatest contribution to sentiment values, in either a positive or negative direction

The bigrams “not like” and “not help” were overwhelmingly the largest causes of misidentification, making the text seem much more positive than it is. But we can see phrases like “not afraid” and “not fail” sometimes suggest text is more negative than it is.

“Not” isn’t the only term that provides some context for the following word. We could pick four common words (or more) that negate the subsequent term, and use the same joining and counting approach to examine all of them at once.

negation\_words <- c("not", "no", "never", "without")

negated\_words <- bigrams\_separated %>%

filter(word1 %in% negation\_words) %>%

inner\_join(AFINN, by = c(word2 = "word")) %>%

count(word1, word2, value, sort = TRUE)

We could then visualize what the most common words to follow each particular negation are (Figure 4.3). While “not like” and “not help” are still the two most common examples, we can also see pairings such as “no great” and “never loved.” We could combine this with the approaches in Chapter 2 to reverse the AFINN values of each word that follows a negation. These are just a few examples of how finding consecutive words can give context to text mining methods.

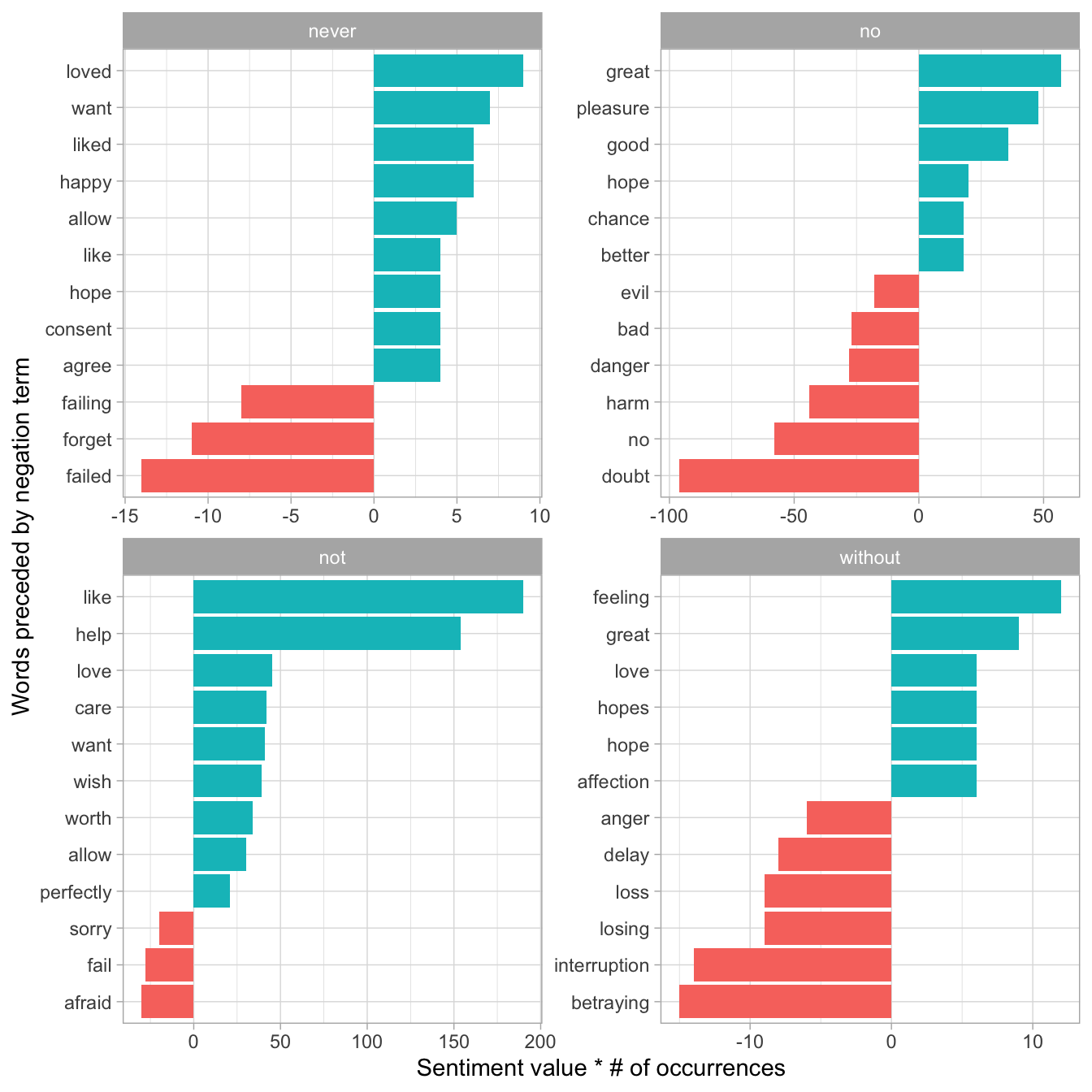


Figure 4.3: Most common positive or negative words to follow negations such as ‘never’, ‘no’, ‘not’, and ‘without’

### 4.1.4 Visualizing a network of bigrams with ggraph

We may be interested in visualizing all of the relationships among words simultaneously, rather than just the top few at a time. As one common visualization, we can arrange the words into a network, or “graph.” Here we’ll be referring to a “graph” not in the sense of a visualization, but as a combination of connected nodes. A graph can be constructed from a tidy object since it has three variables:

* **from**: the node an edge is coming from
* **to**: the node an edge is going towards
* **weight**: A numeric value associated with each edge

The igraph package has many powerful functions for manipulating and analyzing networks. One way to create an igraph object from tidy data is the graph\_from\_data\_frame() function, which takes a data frame of edges with columns for “from”, “to”, and edge attributes (in this case n):

library(igraph)

# original counts

bigram\_counts

#> # A tibble: 28,975 × 3

#> word1 word2 n

#> <chr> <chr> <int>

#> 1 <NA> <NA> 12242

#> 2 sir thomas 266

#> 3 miss crawford 196

#> 4 captain wentworth 143

#> 5 miss woodhouse 143

#> 6 frank churchill 114

#> 7 lady russell 110

#> 8 sir walter 108

#> 9 lady bertram 101

#> 10 miss fairfax 98

#> # … with 28,965 more rows

# filter for only relatively common combinations

bigram\_graph <- bigram\_counts %>%

filter(n > 20) %>%

graph\_from\_data\_frame()

bigram\_graph

#> IGRAPH 8dcffee DN-- 86 71 --

#> + attr: name (v/c), n (e/n)

#> + edges from 8dcffee (vertex names):

#> [1] NA ->NA sir ->thomas miss ->crawford

#> [4] captain ->wentworth miss ->woodhouse frank ->churchill

#> [7] lady ->russell sir ->walter lady ->bertram

#> [10] miss ->fairfax colonel ->brandon sir ->john

#> [13] miss ->bates jane ->fairfax lady ->catherine

#> [16] lady ->middleton miss ->tilney miss ->bingley

#> [19] thousand->pounds miss ->dashwood dear ->miss

#> [22] miss ->bennet miss ->morland captain ->benwick

#> + ... omitted several edges

igraph has plotting functions built in, but they’re not what the package is designed to do, so many other packages have developed visualization methods for graph objects. We recommend the ggraph package (Pedersen 2017), because it implements these visualizations in terms of the grammar of graphics, which we are already familiar with from ggplot2.

We can convert an igraph object into a ggraph with the ggraph function, after which we add layers to it, much like layers are added in ggplot2. For example, for a basic graph we need to add three layers: nodes, edges, and text.

library(ggraph)

set.seed(2017)

ggraph(bigram\_graph, layout = "fr") +

geom\_edge\_link() +

geom\_node\_point() +

geom\_node\_text(aes(label = name), vjust = 1, hjust = 1)

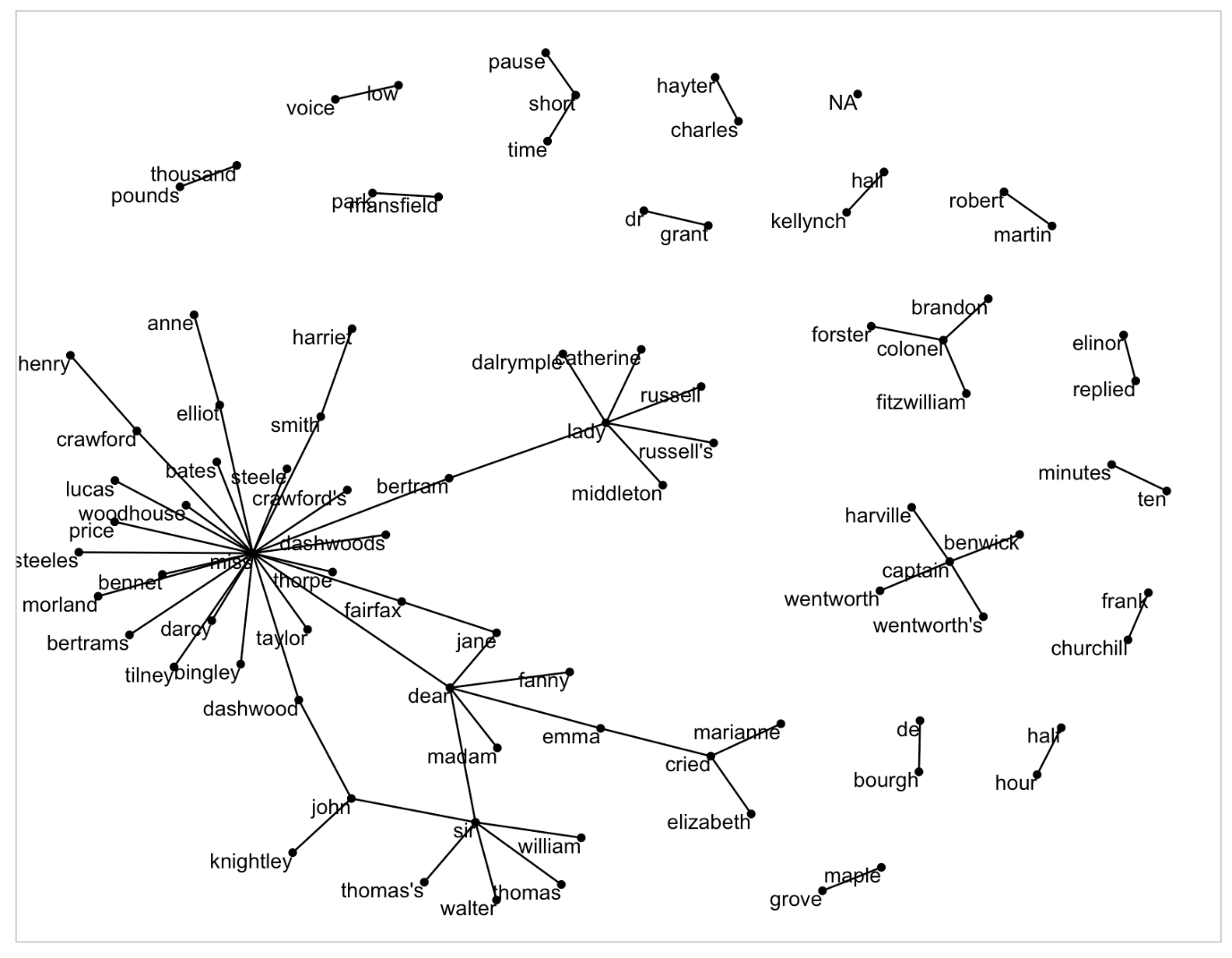


Figure 4.4: Common bigrams in Jane Austen’s novels, showing those that occurred more than 20 times and where neither word was a stop word

In Figure 4.4, we can visualize some details of the text structure. For example, we see that salutations such as “miss”, “lady”, “sir”, and “colonel” form common centers of nodes, which are often followed by names. We also see pairs or triplets along the outside that form common short phrases (“half hour”, “thousand pounds”, or “short time/pause”).

We conclude with a few polishing operations to make a better looking graph (Figure 4.5):

* We add the edge\_alpha aesthetic to the link layer to make links transparent based on how common or rare the bigram is
* We add directionality with an arrow, constructed using grid::arrow(), including an end\_cap option that tells the arrow to end before touching the node
* We tinker with the options to the node layer to make the nodes more attractive (larger, blue points)
* We add a theme that’s useful for plotting networks, theme\_void()

set.seed(2020)

a <- grid::arrow(type = "closed", length = unit(.15, "inches"))

ggraph(bigram\_graph, layout = "fr") +

geom\_edge\_link(aes(edge\_alpha = n), show.legend = FALSE,

arrow = a, end\_cap = circle(.07, 'inches')) +

geom\_node\_point(color = "lightblue", size = 5) +

geom\_node\_text(aes(label = name), vjust = 1, hjust = 1) +

theme\_void()

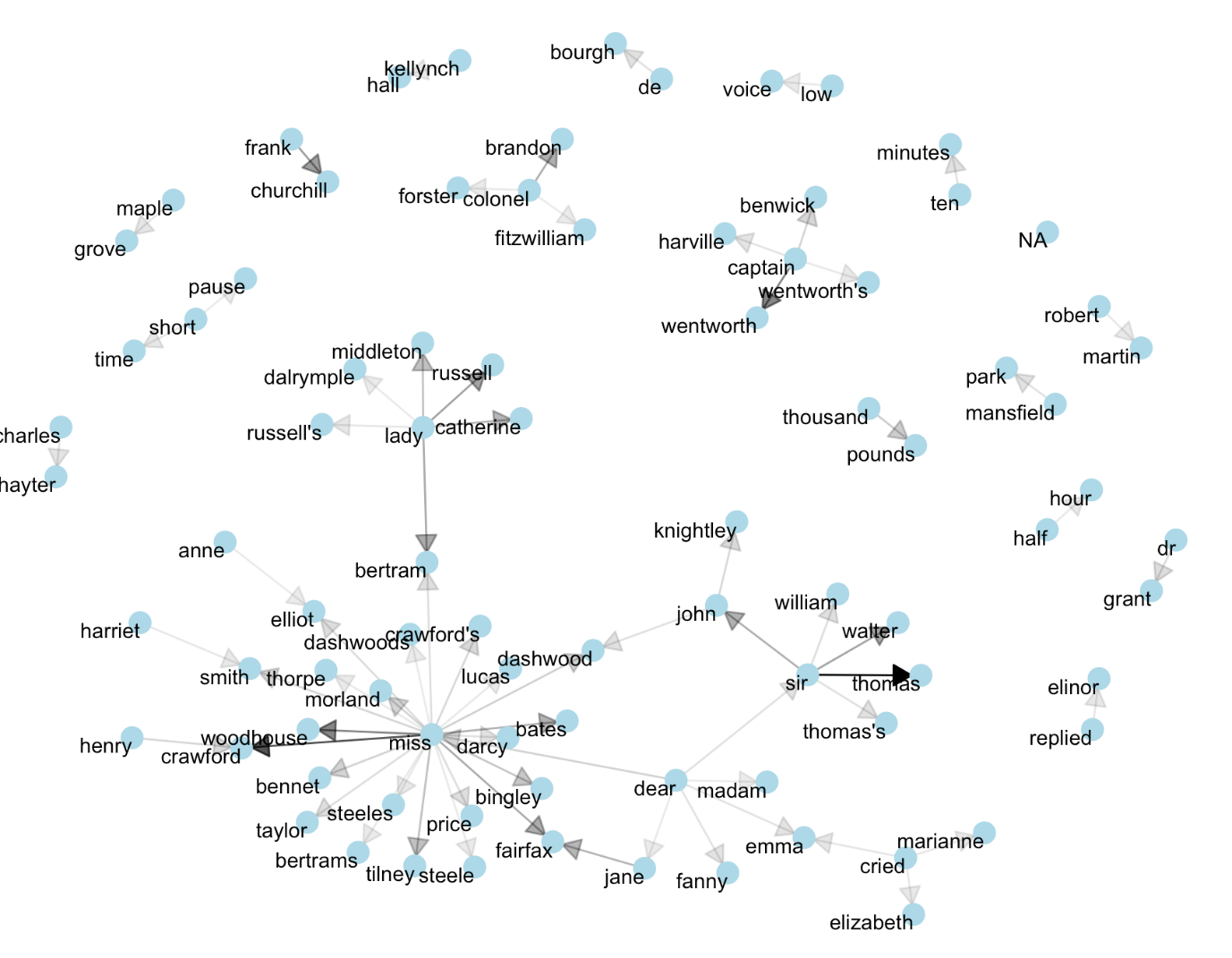


Figure 4.5: Common bigrams in Jane Austen’s novels, with some polishing

It may take some experimentation with ggraph to get your networks into a presentable format like this, but the network structure is useful and flexible way to visualize relational tidy data.

Note that this is a visualization of a **Markov chain**, a common model in text processing. In a Markov chain, each choice of word depends only on the previous word. In this case, a random generator following this model might spit out “dear”, then “sir”, then “william/walter/thomas/thomas’s”, by following each word to the most common words that follow it. To make the visualization interpretable, we chose to show only the most common word to word connections, but one could imagine an enormous graph representing all connections that occur in the text.

### 4.1.5 Visualizing bigrams in other texts

We went to a good amount of work in cleaning and visualizing bigrams on a text dataset, so let’s collect it into a function so that we easily perform it on other text datasets.

To make it easy to use the count\_bigrams() and visualize\_bigrams() yourself, we’ve also reloaded the packages necessary for them.

library(dplyr)

library(tidyr)

library(tidytext)

library(ggplot2)

library(igraph)

library(ggraph)

count\_bigrams <- function(dataset) {

dataset %>%

unnest\_tokens(bigram, text, token = "ngrams", n = 2) %>%

separate(bigram, c("word1", "word2"), sep = " ") %>%

filter(!word1 %in% stop\_words$word,

!word2 %in% stop\_words$word) %>%

count(word1, word2, sort = TRUE)

}

visualize\_bigrams <- function(bigrams) {

set.seed(2016)

a <- grid::arrow(type = "closed", length = unit(.15, "inches"))

bigrams %>%

graph\_from\_data\_frame() %>%

ggraph(layout = "fr") +

geom\_edge\_link(aes(edge\_alpha = n), show.legend = FALSE, arrow = a) +

geom\_node\_point(color = "lightblue", size = 5) +

geom\_node\_text(aes(label = name), vjust = 1, hjust = 1) +

theme\_void()

}

At this point, we could visualize bigrams in other works, such as the King James Version of the Bible:

# the King James version is book 10 on Project Gutenberg:

library(gutenbergr)

kjv <- gutenberg\_download(10)

library(stringr)

kjv\_bigrams <- kjv %>%

count\_bigrams()

# filter out rare combinations, as well as digits

kjv\_bigrams %>%

filter(n > 40,

!str\_detect(word1, "\\d"),

!str\_detect(word2, "\\d")) %>%

visualize\_bigrams()

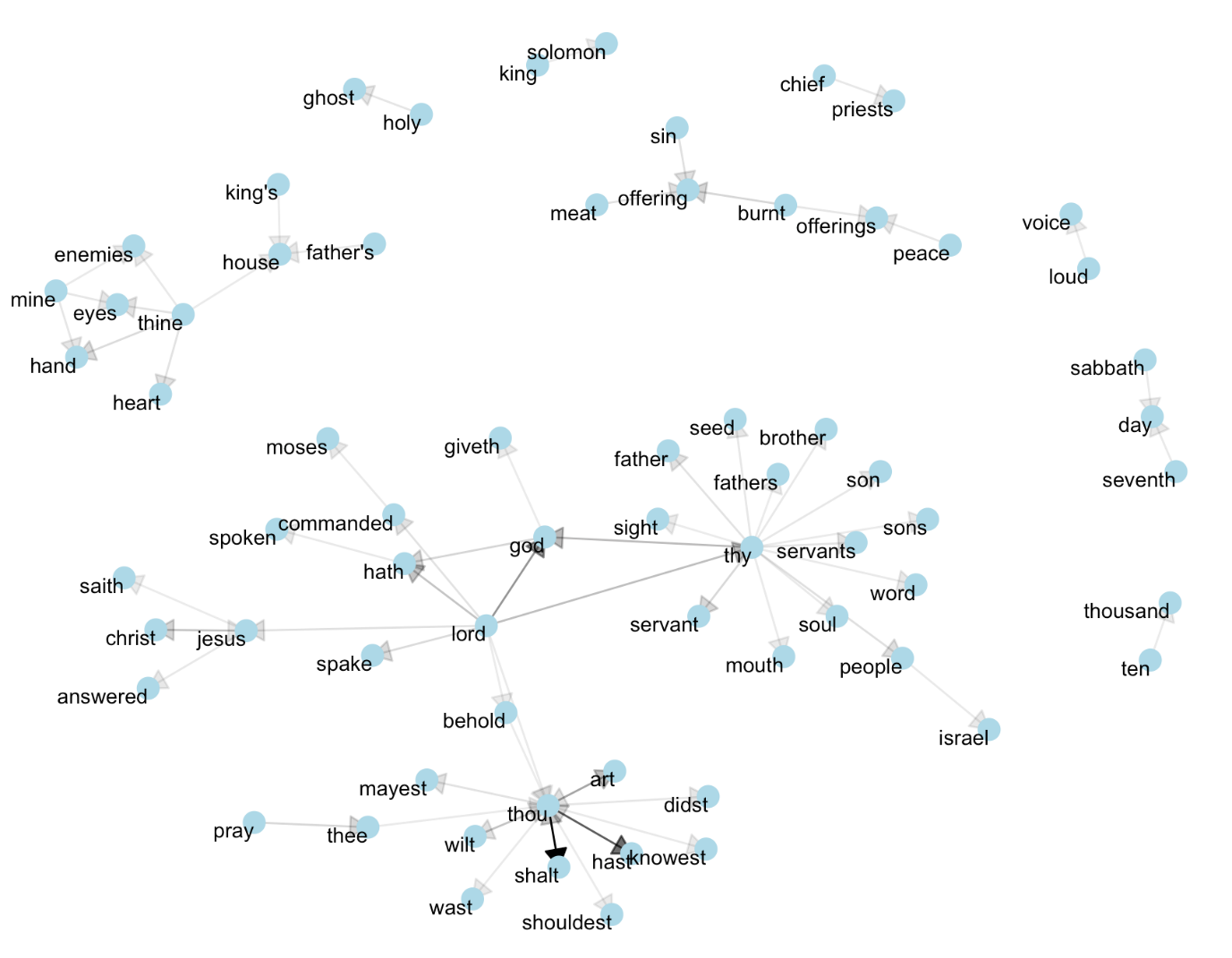


Figure 4.6: Directed graph of common bigrams in the King James Bible, showing those that occurred more than 40 times

Figure 4.6 thus lays out a common “blueprint” of language within the Bible, particularly focused around “thy” and “thou” (which could probably be considered stopwords!) You can use the gutenbergr package and these count\_bigrams/visualize\_bigrams functions to visualize bigrams in other classic books you’re interested in.

## 4.2 Counting and correlating pairs of words with the widyr package

Tokenizing by n-gram is a useful way to explore pairs of adjacent words. However, we may also be interested in words that tend to co-occur within particular documents or particular chapters, even if they don’t occur next to each other.

Tidy data is a useful structure for comparing between variables or grouping by rows, but it can be challenging to compare between rows: for example, to count the number of times that two words appear within the same document, or to see how correlated they are. Most operations for finding pairwise counts or correlations need to turn the data into a wide matrix first.

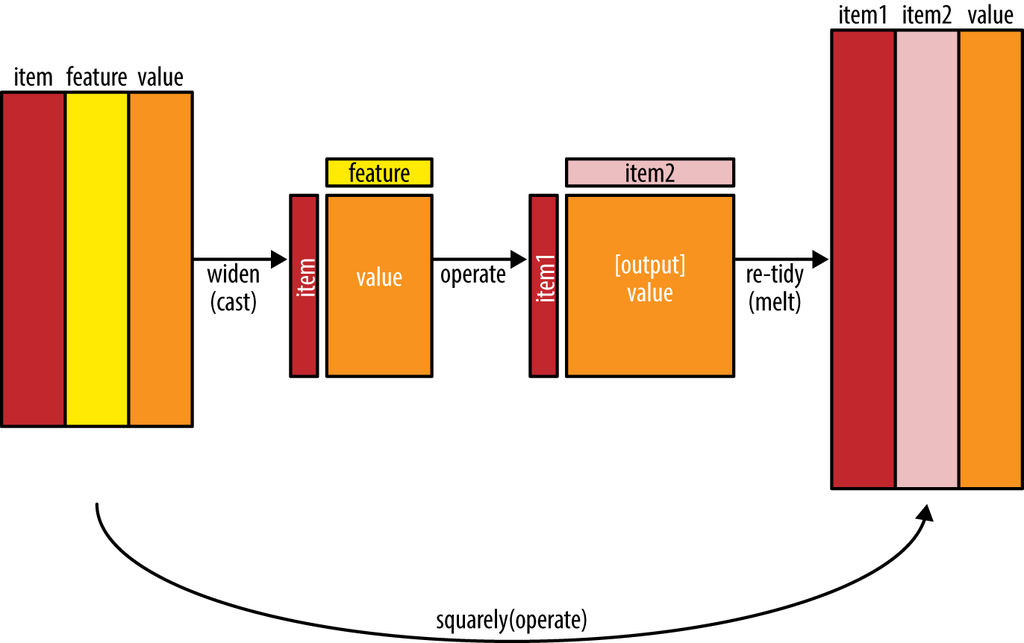


Figure 4.7: The philosophy behind the widyr package, which can perform operations such as counting and correlating on pairs of values in a tidy dataset. The widyr package first ‘casts’ a tidy dataset into a wide matrix, performs an operation such as a correlation on it, then re-tidies the result.

We’ll examine some of the ways tidy text can be turned into a wide matrix in Chapter 5, but in this case it isn’t necessary. The widyr package makes operations such as computing counts and correlations easy, by simplifying the pattern of “widen data, perform an operation, then re-tidy data” (Figure 4.7). We’ll focus on a set of functions that make pairwise comparisons between groups of observations (for example, between documents, or sections of text).

### 4.2.1 Counting and correlating among sections

Consider the book “Pride and Prejudice” divided into 10-line sections, as we did (with larger sections) for sentiment analysis in Chapter 2. We may be interested in what words tend to appear within the same section.

austen\_section\_words <- austen\_books() %>%

filter(book == "Pride & Prejudice") %>%

mutate(section = row\_number() %/% 10) %>%

filter(section > 0) %>%

unnest\_tokens(word, text) %>%

filter(!word %in% stop\_words$word)

austen\_section\_words

#> # A tibble: 37,240 × 3

#> book section word

#> <fct> <dbl> <chr>

#> 1 Pride & Prejudice 1 truth

#> 2 Pride & Prejudice 1 universally

#> 3 Pride & Prejudice 1 acknowledged

#> 4 Pride & Prejudice 1 single

#> 5 Pride & Prejudice 1 possession

#> 6 Pride & Prejudice 1 fortune

#> 7 Pride & Prejudice 1 wife

#> 8 Pride & Prejudice 1 feelings

#> 9 Pride & Prejudice 1 views

#> 10 Pride & Prejudice 1 entering

#> # … with 37,230 more rows

One useful function from widyr is the pairwise\_count() function. The prefix pairwise\_ means it will result in one row for each pair of words in the word variable. This lets us count common pairs of words co-appearing within the same section:

library(widyr)

# count words co-occuring within sections

word\_pairs <- austen\_section\_words %>%

pairwise\_count(word, section, sort = TRUE)

word\_pairs

#> # A tibble: 796,008 × 3

#> item1 item2 n

#> <chr> <chr> <dbl>

#> 1 darcy elizabeth 144

#> 2 elizabeth darcy 144

#> 3 miss elizabeth 110

#> 4 elizabeth miss 110

#> 5 elizabeth jane 106

#> 6 jane elizabeth 106

#> 7 miss darcy 92

#> 8 darcy miss 92

#> 9 elizabeth bingley 91

#> 10 bingley elizabeth 91

#> # … with 795,998 more rows

Notice that while the input had one row for each pair of a document (a 10-line section) and a word, the output has one row for each pair of words. This is also a tidy format, but of a very different structure that we can use to answer new questions.

For example, we can see that the most common pair of words in a section is “Elizabeth” and “Darcy” (the two main characters). We can easily find the words that most often occur with Darcy:

word\_pairs %>%

filter(item1 == "darcy")

#> # A tibble: 2,930 × 3

#> item1 item2 n

#> <chr> <chr> <dbl>

#> 1 darcy elizabeth 144

#> 2 darcy miss 92

#> 3 darcy bingley 86

#> 4 darcy jane 46

#> 5 darcy bennet 45

#> 6 darcy sister 45

#> 7 darcy time 41

#> 8 darcy lady 38

#> 9 darcy friend 37

#> 10 darcy wickham 37

#> # … with 2,920 more rows

### 4.2.2 Pairwise correlation

Pairs like “Elizabeth” and “Darcy” are the most common co-occurring words, but that’s not particularly meaningful since they’re also the most common individual words. We may instead want to examine **correlation** among words, which indicates how often they appear together relative to how often they appear separately.

In particular, here we’ll focus on the phi coefficient, a common measure for binary correlation. The focus of the phi coefficient is how much more likely it is that either **both** word X and Y appear, or **neither** do, than that one appears without the other.

Consider the following table:

|  | **Has word Y** | **No word Y** | **Total** |  |
| --- | --- | --- | --- | --- |
| Has word X | n11n11 | n10n10 | n1⋅n1⋅ |  |
| No word X | n01n01 | n00n00 | n0⋅n0⋅ |  |
| Total | n⋅1n⋅1 | n⋅0n⋅0 | n |  |

For example, that n11n11 represents the number of documents where both word X and word Y appear, n00n00 the number where neither appears, and n10n10 and n01n01 the cases where one appears without the other. In terms of this table, the phi coefficient is:

ϕ=n11n00−n10n01√n1⋅n0⋅n⋅0n⋅1ϕ=n11n00−n10n01n1⋅n0⋅n⋅0n⋅1

The phi coefficient is equivalent to the Pearson correlation, which you may have heard of elsewhere when it is applied to binary data.

The pairwise\_cor() function in widyr lets us find the phi coefficient between words based on how often they appear in the same section. Its syntax is similar to pairwise\_count().

# we need to filter for at least relatively common words first

word\_cors <- austen\_section\_words %>%

group\_by(word) %>%

filter(n() >= 20) %>%

pairwise\_cor(word, section, sort = TRUE)

word\_cors

#> # A tibble: 154,842 × 3

#> item1 item2 correlation

#> <chr> <chr> <dbl>

#> 1 bourgh de 0.951

#> 2 de bourgh 0.951

#> 3 pounds thousand 0.701

#> 4 thousand pounds 0.701

#> 5 william sir 0.664

#> 6 sir william 0.664

#> 7 catherine lady 0.663

#> 8 lady catherine 0.663

#> 9 forster colonel 0.622

#> 10 colonel forster 0.622

#> # … with 154,832 more rows

This output format is helpful for exploration. For example, we could find the words most correlated with a word like “pounds” using a filter operation.

word\_cors %>%

filter(item1 == "pounds")

#> # A tibble: 393 × 3

#> item1 item2 correlation

#> <chr> <chr> <dbl>

#> 1 pounds thousand 0.701

#> 2 pounds ten 0.231

#> 3 pounds fortune 0.164

#> 4 pounds settled 0.149

#> 5 pounds wickham's 0.142

#> 6 pounds children 0.129

#> 7 pounds mother's 0.119

#> 8 pounds believed 0.0932

#> 9 pounds estate 0.0890

#> 10 pounds ready 0.0860

#> # … with 383 more rows

This lets us pick particular interesting words and find the other words most associated with them (Figure 4.8).

word\_cors %>%

filter(item1 %in% c("elizabeth", "pounds", "married", "pride")) %>%

group\_by(item1) %>%

slice\_max(correlation, n = 6) %>%

ungroup() %>%

mutate(item2 = reorder(item2, correlation)) %>%

ggplot(aes(item2, correlation)) +

geom\_bar(stat = "identity") +

facet\_wrap(~ item1, scales = "free") +

coord\_flip()

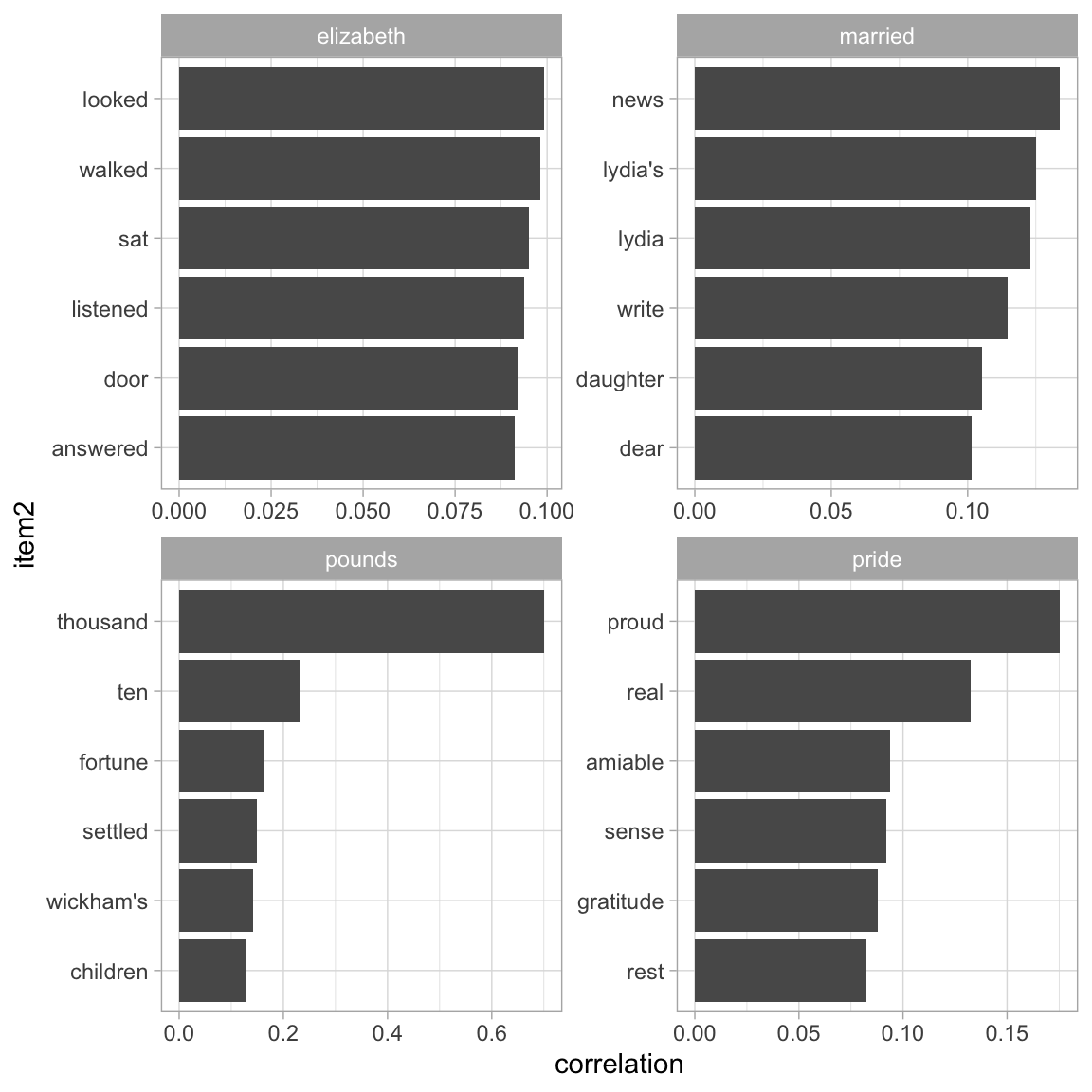


Figure 4.8: Words from Pride and Prejudice that were most correlated with ‘elizabeth’, ‘pounds’, ‘married’, and ‘pride’

Just as we used ggraph to visualize bigrams, we can use it to visualize the correlations and clusters of words that were found by the widyr package (Figure 4.9).

set.seed(2016)

word\_cors %>%

filter(correlation > .15) %>%

graph\_from\_data\_frame() %>%

ggraph(layout = "fr") +

geom\_edge\_link(aes(edge\_alpha = correlation), show.legend = FALSE) +

geom\_node\_point(color = "lightblue", size = 5) +

geom\_node\_text(aes(label = name), repel = TRUE) +

theme\_void()

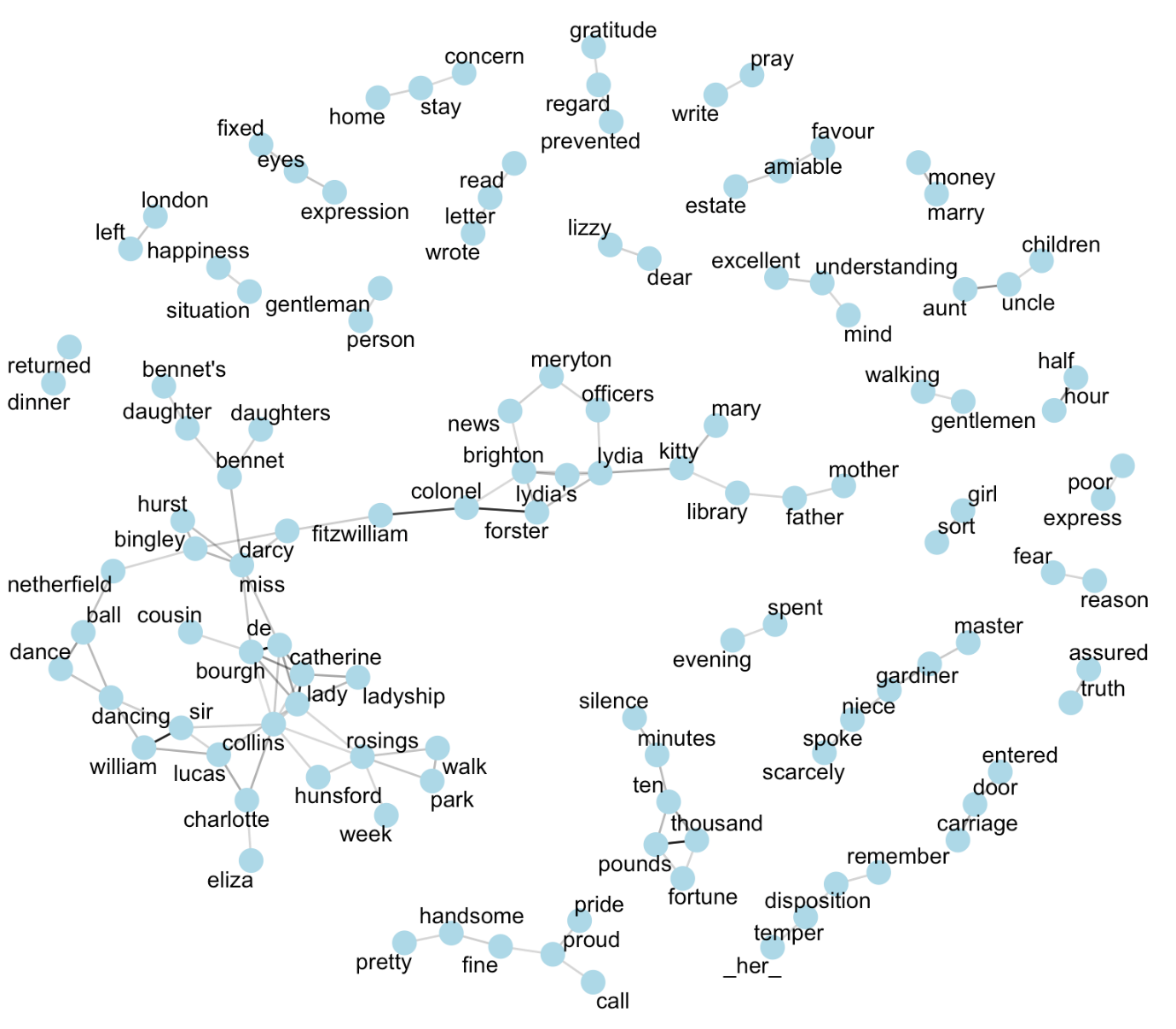


Figure 4.9: Pairs of words in Pride and Prejudice that show at least a .15 correlation of appearing within the same 10-line section

Note that unlike the bigram analysis, the relationships here are symmetrical, rather than directional (there are no arrows). We can also see that while pairings of names and titles that dominated bigram pairings are common, such as “colonel/fitzwilliam”, we can also see pairings of words that appear close to each other, such as “walk” and “park”, or “dance” and “ball”.

we add a condition for eliminating words mentioned in only one  
submission, often function names.

users <- unique(threads$user)

onboarding\_line\_words <- threads %>%

dplyr::group\_by(user, issue, created\_at, package, line) %>%

dplyr::mutate(line\_id = paste(package, user, created\_at, line)) %>%

dplyr::ungroup() %>%

tidytext::unnest\_tokens(word, line) %>%

dplyr::filter( word != package, !word %in% users,

is.na(as.numeric(word)),

word != "ldecicco",

word != "usgs") %>%

dplyr::group\_by(word) %>%

dplyr::filter(length(unique(issue)) > 1) %>%

dplyr::select(line\_id, word)

onboarding\_line\_words %>%

head() %>%

knitr::kable()

| **line\_id** | **word** |
| --- | --- |
| rrlite karthik 2015-04-12 20:56:04 – ] add a ropensci footer. | add |
| rrlite karthik 2015-04-12 20:56:04 – ] add a ropensci footer. | a |
| rrlite karthik 2015-04-12 20:56:04 – ] add a ropensci footer. | ropensci |
| rrlite karthik 2015-04-12 20:56:04 – ] add a ropensci footer. | footer |
| rrlite karthik 2015-04-12 20:56:04 – ] add an appropriate entry into ropensci.org/packages/index.html | add |
| rrlite karthik 2015-04-12 20:56:04 – ] add an appropriate entry into ropensci.org/packages/index.html | an |
|  |  |

Then, we can compute the correlation.

word\_cors <- onboarding\_line\_words %>%

dplyr::group\_by(word) %>%

dplyr::filter(!word %in% stopwords) %>%

dplyr::filter(n() >= 20) %>%

widyr::pairwise\_cor(word, line\_id, sort = TRUE)

For instance, what often goes in the same line as vignette?

dplyr::filter(word\_cors, item1 == "vignette")

## # A tibble: 853 x 3

## item1 item2 correlation

##

## 1 vignette readme 0.176

## 2 vignette vignettes 0.174

## 3 vignette chunk 0.145

## 4 vignette eval 0.120

## 5 vignette examples 0.108

## 6 vignette overview 0.0933

## 7 vignette building 0.0914

## 8 vignette link 0.0863

## 9 vignette maps 0.0840

## 10 vignette package 0.0831

## # ... with 843 more rows

Now let’s plot the network of these relationships between words, using  
the igraph package and ggraph .

library("igraph")

library("ggraph")

set.seed(2016)

word\_cors %>%

dplyr::filter(correlation > .35) %>%

graph\_from\_data\_frame() %>%

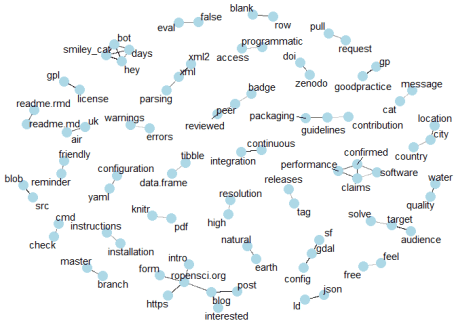
ggraph(layout = "fr") +

geom\_edge\_link(aes(edge\_alpha = correlation), show.legend = FALSE) +

geom\_node\_point(color = "lightblue", size = 5) +

geom\_node\_text(aes(label = name), repel = TRUE) +

theme\_void()



This figure gives a good sample of things discussed in reviews. Despite  
our efforts filtering words specific to issues, some of them remain very  
specific, such as country/city/location that are very frequent in  
ropenaq review.

**How positive is onboarding?**

Using sentiment analysis, we can look at how positive comments are.

sentiments %>%

dplyr::group\_by(role) %>%

skimr::skim(sentiment)

## Skim summary statistics

## n obs: 11553

## n variables: 6

## group variables: role

##

## Variable type: numeric

## role variable missing complete n mean sd min p25

## author sentiment 0 4823 4823 0.07 0.21 -1.2 0

## community\_manager sentiment 0 97 97 0.13 0.21 -0.41 0

## editor sentiment 0 1521 1521 0.13 0.22 -1.63 0

## other sentiment 0 344 344 0.073 0.2 -0.6 0

## reviewer sentiment 0 4768 4768 0.073 0.21 -1 0

## median p75 max hist

## 0 0.17 1.84

## 0.071 0.23 1

## 0.075 0.25 1.13

## 0 0.2 0.81

## 0 0.17 1.73

summary(sentiments$sentiment)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## -1.63200 0.00000 0.00000 0.07961 0.18353 1.84223

sentiments %>%

dplyr::filter(!role %in% c("other", "community\_manager")) %>%

ggplot(aes(role, sentiment)) +

geom\_boxplot(fill = "salmon") +

hrbrthemes::theme\_ipsum(base\_size = 16,

axis\_title\_size = 16,

strip\_text\_size = 16)

Sentiment of onboarding review threads by
line

These boxplots seem to indicate that lines are generally positive  
(positive mean, zero 25th-quantile), although it’d be better to be able  
to compare them with text from traditional review processes of  
scientific manuscripts in order to get a better feeling for the meaning  
of these numbers.

On these boxplots we also see that we do get lines with a negative  
sentiment value… about what? Here are the most common words in negative  
lines.

sentiments %>%

dplyr::filter(sentiment < 0) %>%

tidytext::unnest\_tokens(word, line) %>%

dplyr::filter(!word %in% stopwords) %>%

dplyr::count(word, sort = TRUE) %>%

dplyr::mutate(word = reorder(word, n)) %>%

dplyr::filter(n > 100) %>%

ggplot() +

geom\_lollipop(aes(word, n),

size = 2, col = "salmon") +

hrbrthemes::theme\_ipsum(base\_size = 16,

axis\_title\_size = 16) +

coord\_flip()

Most common words in negative
lines

And looking at a sample…

sentiments %>%

dplyr::arrange(sentiment) %>%

dplyr::select(line, sentiment) %>%

head(n = 15) %>%

knitr::kable()

| **line** | **sentiment** |
| --- | --- |
| @ultinomics no more things, although do make sure to add more examples – perhaps open an issue ropenscilabs/gtfsr/issues to remind yourself to do that, | -1.6320000 |
| not sure what you mean, but i’ll use different object names to avoid any confusion (ropenscilabs/mregions#24) | -1.2029767 |
| error in .local(.object, …) : | -1.0000000 |
| error: | -1.0000000 |
| #### miscellaneous | -1.0000000 |
| error: command failed (1) | -0.8660254 |
| – get\_plate\_size\_from\_number\_of\_columns: maybe throwing an error makes more sense than returning a string indicating an error | -0.7855844 |
| this code returns an error, which is good, but it would be useful to return a more clear error. filtering on a non-existant species results in a 0 “length” onekp object (ok), but then the download\_\* functions return a curl error due to a misspecified url. | -0.7437258 |
| 0 errors | 0 warnings | 0 notes | -0.7216878 |
| once i get to use this package more, i’m sure i’ll have more comments/issues but for the moment i just want to get this review done so it isn’t a blocker. | -0.7212489 |
| – i now realize i’ve pasted the spelling mistakes without thinking too much about us vs. uk english, sorry. | -0.7071068 |
| minor issues: | -0.7071068 |
| ## minor issues | -0.7071068 |
| replicates issue | -0.7071068 |
| visualization issue | -0.7071068 |
|  |  |

It seems that negative lines are mostly people discussing bugs and  
problems in code, and GitHub issues, and trying to solve them. The kind  
of negative lines we’re happy to see in our process, since once solved,  
they mean the software got more robust!

Last but not least, I mentioned our using particular cases as examples  
of how happy everyone seems to be in the process. To find such examples,  
we rely on memory, but what about picking heart-warming lines using  
their sentiment score?

sentiments %>%

dplyr::arrange(- sentiment) %>%

dplyr::select(line, sentiment) %>%

head(n = 15) %>%

knitr::kable()

| **line** | **sentiment** |
| --- | --- |
| absolutely – it’s really important to ensure it really has been solved! | 1.842234 |
| overall, really easy to use and really nicely done. | 1.733333 |
| this package is a great and lightweight addition to working with rdf and linked data in r. coming after my review of the codemetar package which introduced me to linked data, i found this a great learning experience into a topic i’ve become really interested in but am still quite novice in so i hope my feedback helps to appreciate that particular pov. | 1.463226 |
| i am very grateful for your approval and i very much look forward to collaborating with you and the ropensci community. | 1.256935 |
| thank you very much for the constructive thoughts. | 1.237437 |
| thanks for the approval, all in all a very helpful and educational process! | 1.217567 |
| – really good use of helper functions | 1.139013 |
| – i believe the utf note is handled correctly and this is just a snafu in **goodpractice**, but i will seek a reviewer with related expertise in ensuring that all unicode is handled properly. | 1.132201 |
| seem more unified and consistent. | 1.126978 |
| very much appreciated! | 1.125833 |
| – well organized, readable code | 1.100000 |
| – wow very extensive testing! well done, very thorough | 1.100000 |
| – i’m delighted that you find my work interesting and i’m very keen to help, contribute and collaborate in any capacity. | 1.084493 |
| thank you very much for your thorough and thoughtful review, @batpigandme ! this is great feedback, and i think that visdat will be much improved because of these reviews. | 1.083653 |
| great, thank you very much for accepting this package. i am very grateful about the reviews, which were very helpful to improve this package! | 1.074281 |
|  |  |

As you can imagine, these sentences make the whole team very happy! And  
we hope they’ll encourage you to contribute to rOpenSci onboarding.

**Outlook**

This first try at text analysis of onboarding issue threads is quite  
promising: we were able to retrieve text and to use natural language  
processing to extract most common words and bigrams, and sentiment. This  
allowed us to describe the social weather of onboarding.