Introduction to Machine Learning in R

Lab 3: Text as Data

Table of contents

Note

This script consists of excerpts from Julia Silge and David Robinson's fantastic book Text Mining with R: A Tidy Approach. Check out the online book for more examples and additional information: https://www.tidytextmining.com/.

Load Required Packages

```
library(here)
library(tidyverse)
library(tidytext)
library(textdata)
library(topicmodels)
library(janeaustenr)
```

1. Working with text data

The unnest_tokens function

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 -"
[2] "He kindly stopped for me -"
[3] "The Carriage held but just Ourselves -"
[4] "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 <- tibble(line = 1:4, text = text)

text_df
```

```
# A tibble: 4 x 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, though. 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.

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.

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 tidytext's `unnest_tokens()` function.

```
library(tidytext)

text_df %>%
  unnest_tokens(word, text)
```

```
# A tibble: 20 x 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
11     2 for
12     2 me
```

```
13     3 the
14     3 carriage
15     3 held
16     3 but
17     3 just
18     3 ourselves
19     4 and
20     4 immortality
```

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 = FALSE` argument to turn off this behavior).

Tidying the works of Jane Austen

Let's use the text of Jane Austen's 6 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.

```
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 x 4
   text                    book                linenumber chapter
   <chr>                   <fct>                    <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
# i 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.

```
tidy_books <- original_books %>%
  unnest_tokens(word, text)

tidy_books
```

```
# A tibble: 725,055 x 4
   book                linenumber chapter word
   <fct>                    <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
# i 725,045 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; stop words 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)
```

```
Joining with `by = join_by(word)`
```

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.

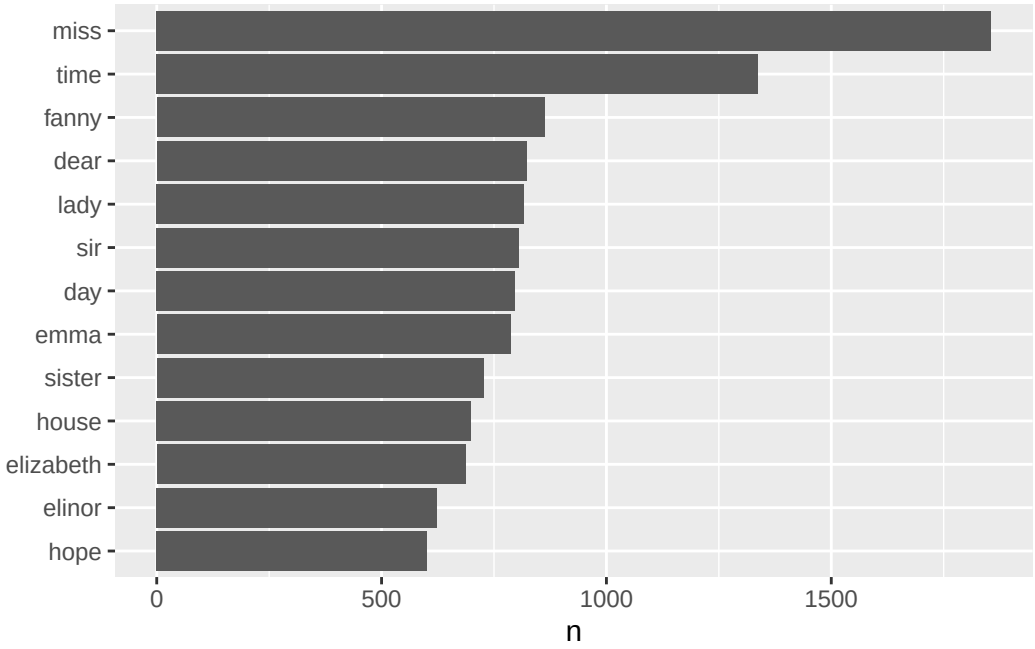
We can also use dplyr's `count()` to find the most common words in all the books as a whole.

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

```
# A tibble: 13,914 x 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
# i 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 this directly to the ggplot2 package, for example to create a visualization of the most common words.

```r
tidy_books %>%
  count(word, sort = TRUE) %>%
  filter(n > 600) %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(n, word)) +
  geom_col() +
  labs(y = NULL)
```



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.

2. Sentiment Analysis

The `sentiments` datasets

There are a variety of methods and dictionaries that exist for evaluating the opinion or emotion in text. The `tidytext` package provides access to several sentiment lexicons. Three general-purpose lexicons are

- `AFINN` from Finn Årup Nielsen,
- `bing` from Bing Liu and collaborators, and

- `nrc` from Saif Mohammad and Peter Turney.

All three of these 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.

These lexicons are available under different licenses, so be sure that the license for the lexicon you want to use is appropriate for your project. You may be asked to agree to a license before downloading data.

The function `get_sentiments()` allows us to get specific sentiment lexicons with the appropriate measures for each one.

```
get_sentiments("afinn")
```

```
# A tibble: 2,477 x 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
# i 2,467 more rows
```

```
get_sentiments("bing")
```

```
# A tibble: 6,786 x 2
   word        sentiment
   <chr>       <chr>
 1 2-faces     negative
 2 abnormal    negative
```

```
 3 abolish     negative
 4 abominable  negative
 5 abominably  negative
 6 abominate   negative
 7 abomination negative
 8 abort       negative
 9 aborted     negative
10 aborts      negative
# i 6,776 more rows
```

```
get_sentiments("nrc")
```

```
# A tibble: 13,872 x 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
# i 13,862 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 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.

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 not sustained sections of sarcasm or negated text, so this is not an important effect.

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 averaged out to about zero, while sentence-sized or paragraph-sized text often works better.

2.2 Sentiment analysis with inner join

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 antijoin operation, performing sentiment analysis is an inner join operation.

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 novels and convert the text to the tidy format using `unnest_tokens()`. Let's also set up some other columns to keep track 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 [\\divxlc]",
                                      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 books 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`.

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

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

```
Joining with `by = join_by(word)`
```

```
# A tibble: 301 x 2
   word          n
   <chr>     <int>
 1 good        359
 2 friend      166
 3 hope        143
 4 happy       125
 5 love        117
 6 deal         92
 7 found        92
 8 present      89
 9 kind         82
10 happiness    76
# i 291 more rows
```

We see mostly positive, happy words about hope, friendship, and love here. We also see some words that may not be used joyfully by Austen ("found", "present"); we will discuss this in more detail in Section 2.4.

We can also 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 `pivot_wider()` so that we have negative and positive sentiment in separate columns, and lastly calculate a net sentiment (positive - negative).

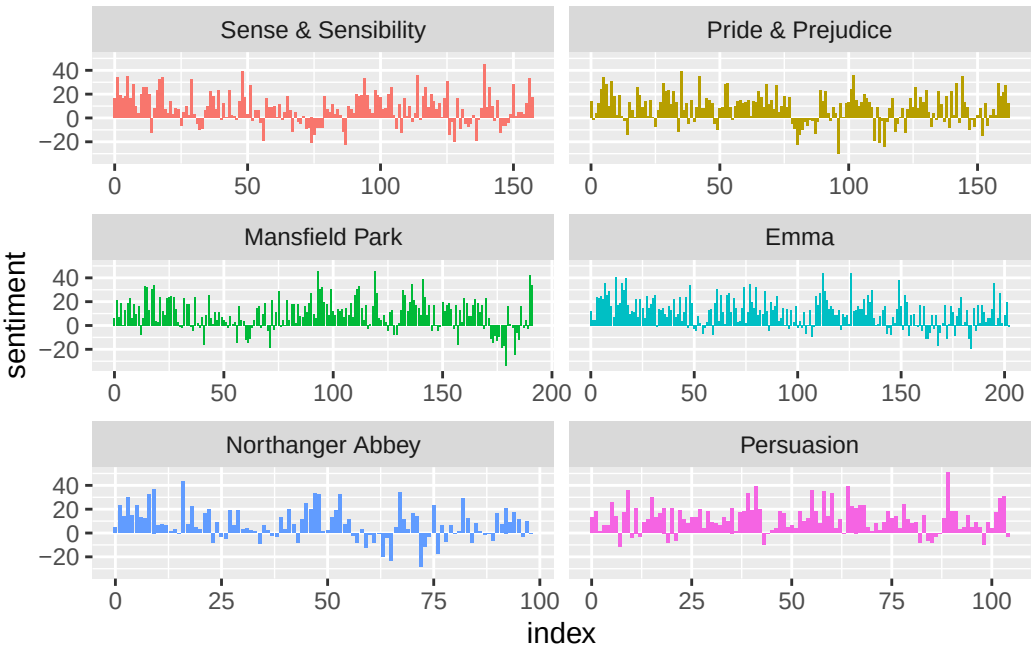
```
library(tidyr)

jane_austen_sentiment <- tidy_books %>%
  inner_join(get_sentiments("bing")) %>%
  count(book, index = linenumber %/% 80, sentiment) %>%
  pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
  mutate(sentiment = positive - negative)
```

```
Joining with `by = join_by(word)`
```

```
Warning in inner_join(., get_sentiments("bing")): Detected an unexpected many-to-many relati
i Row 435434 of `x` matches multiple rows in `y`.
i Row 5051 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
  "many-to-many"` to silence this warning.
```

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.

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



We can see in the Figure 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 Prejudice. 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 x 4
   book              linenumber chapter word
   <fct>                  <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
# i 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.

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()`, `pivot_wider()`, 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(value)) %>%
  mutate(method = "AFINN")
```

```
Joining with `by = join_by(word)`
```

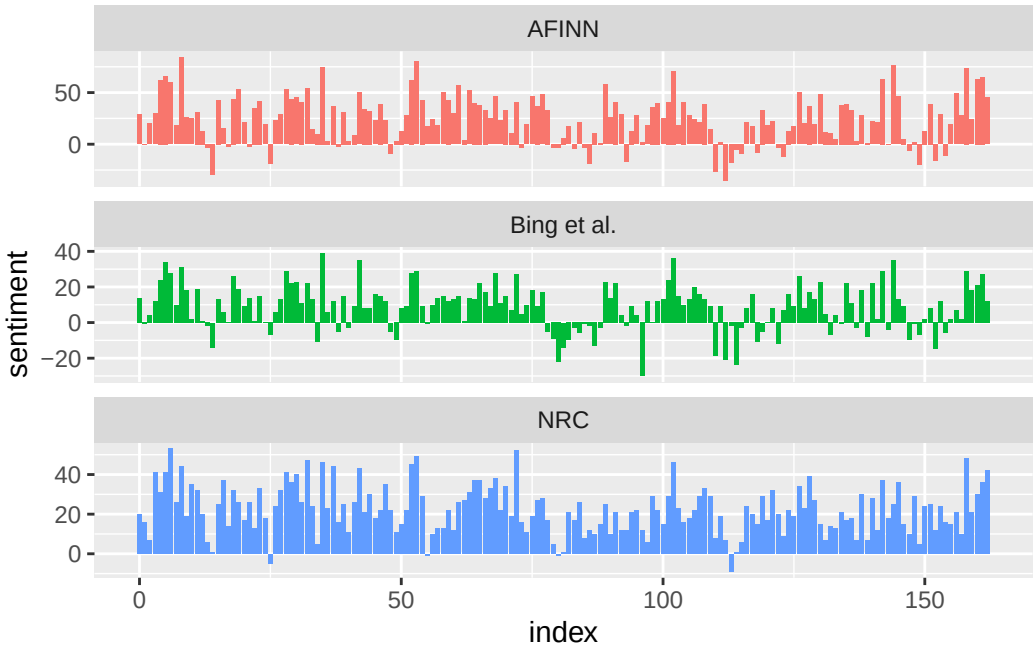
```
bing_and_nrc <- bind_rows(
  pride_prejudice %>%
    inner_join(get_sentiments("bing")) %>%
    mutate(method = "Bing et al."),
  pride_prejudice %>%
    inner_join(get_sentiments("nrc") %>%
                 filter(sentiment %in% c("positive",
                                         "negative"))
    ) %>%
    mutate(method = "NRC")) %>%
  count(method, index = linenumber %/% 80, sentiment) %>%
  pivot_wider(names_from = sentiment,
              values_from = n,
              values_fill = 0) %>%
  mutate(sentiment = positive - negative)
```

```
Joining with `by = join_by(word)`
Joining with `by = join_by(word)`
```

```
Warning in inner_join(., get_sentiments("nrc") %>% filter(sentiment %in% : Detected an unexpe
i Row 215 of `x` matches multiple rows in `y`.
i Row 5178 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
  "many-to-many"` to silence this warning.
```

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.

```
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")
```



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 values 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, 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 x 2
  sentiment     n
  <chr>     <int>
1 negative   3316
2 positive   2308
```

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

```
# A tibble: 2 x 2
  sentiment     n
  <chr>     <int>
1 negative   4781
2 positive   2005
```

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, e.g. if the negative words in the NRC lexicon do not match the words that Jane Austen uses very well. 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 all important context to keep in mind when choosing a sentiment lexicon for analysis.
2.4 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.

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

```
Joining with `by = join_by(word)`
```

```
Warning in inner_join(., get_sentiments("bing")): Detected an unexpected many-to-many relatio
i Row 435434 of `x` matches multiple rows in `y`.
i Row 5051 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
  "many-to-many"` to silence this warning.
```

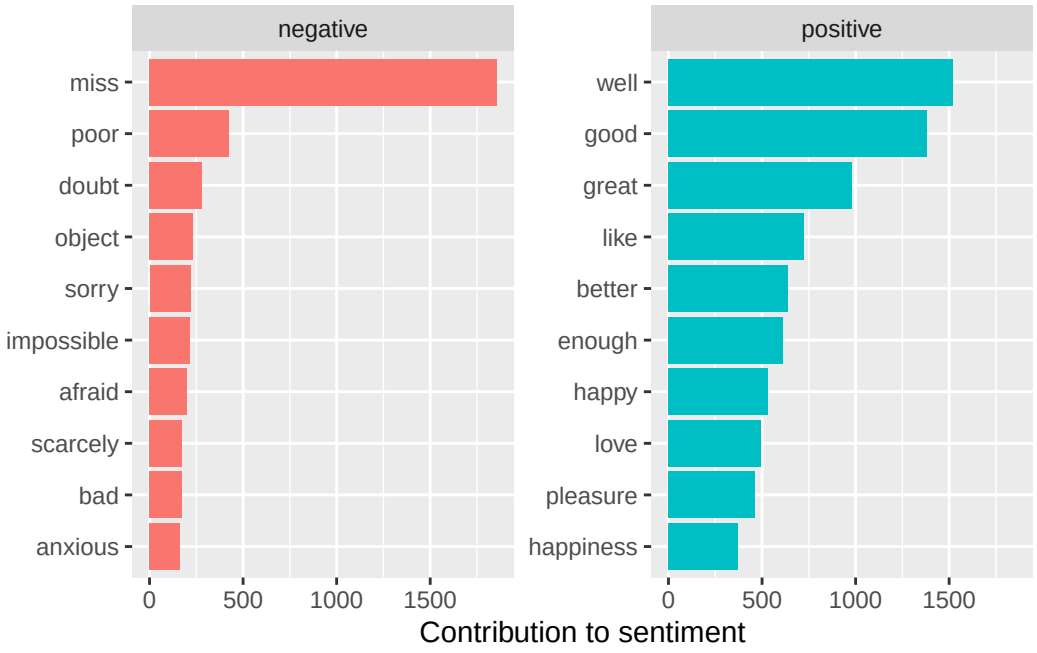
```
bing_word_counts
```

```
# A tibble: 2,585 x 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
# i 2,575 more rows
```

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.

```
bing_word_counts %>%
  group_by(sentiment) %>%
  slice_max(n, n = 10) %>%
  ungroup() %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(n, word, fill = sentiment)) +
  geom_col(show.legend = FALSE) +
```

```
  facet_wrap(~sentiment, scales = "free_y") +
  labs(x = "Contribution to sentiment",
       y = NULL)
```



This figure 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(tibble(word = c("miss"),
                                      lexicon = c("custom")),
                               stop_words)

custom_stop_words
```

```
# A tibble: 1,150 x 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
# i 1,140 more rows
```

3. Topic Modeling

Latent Dirichlet allocation

Latent Dirichlet allocation is one of the most common algorithms for topic modeling. Without diving into the math behind the model, we can understand it as being guided by two principles.

- Every document is a mixture of topics. We imagine that each document may contain words from several topics in particular proportions. For example, in a two-topic model we could say "Document 1 is 90% topic A and 10% topic B, while Document 2 is 30% topic A and 70% topic B."
- Every topic is a mixture of words. For example, we could imagine a two-topic model of American news, with one topic for "politics" and one for "entertainment." The most common words in the politics topic might be "President", "Congress", and "government", while the entertainment topic may be made up of words such as "movies", "television", and "actor". Importantly, words can be shared between topics; a word like "budget" might appear in both equally.

LDA is a mathematical method for estimating both of these at the same time: finding the mixture of words that is associated with each topic, while also determining the mixture of topics that describes each document. There are a number of existing implementations of this algorithm, and we'll explore one of them in depth.

In this example, we use the `AssociatedPress` dataset provided by the `topicmodels` package. This is a collection of 2246 news articles from an American news agency, mostly published around 1988.

```
data("AssociatedPress")
AssociatedPress
```

```
<<DocumentTermMatrix (documents: 2246, terms: 10473)>>
Non-/sparse entries: 302031/23220327
Sparsity           : 99%
Maximal term length: 18
```

```
Weighting          : term frequency (tf)
```

We can use the `LDA()` function from the topicmodels package, setting `k = 2`, to create a two-topic LDA model.

Almost any topic model in practice will use a larger k, but we will soon see that this analysis approach extends to a larger number of topics.

This function returns an object containing the full details of the model fit, such as how words are associated with topics and how topics are associated with documents.

```
# set a seed so that the output of the model is predictable
ap_lda <- LDA(AssociatedPress, k = 2, control = list(seed = 1234))
ap_lda
```

```
A LDA_VEM topic model with 2 topics.
```

Fitting the model was the "easy part": the rest of the analysis will involve exploring and interpreting the model using tidying functions from the `tidytext` package.

Word-topic probabilities

The `tidytext` package provides the `tidy()` method for extracting the per-topic-per-word probabilities, called β, from the model.

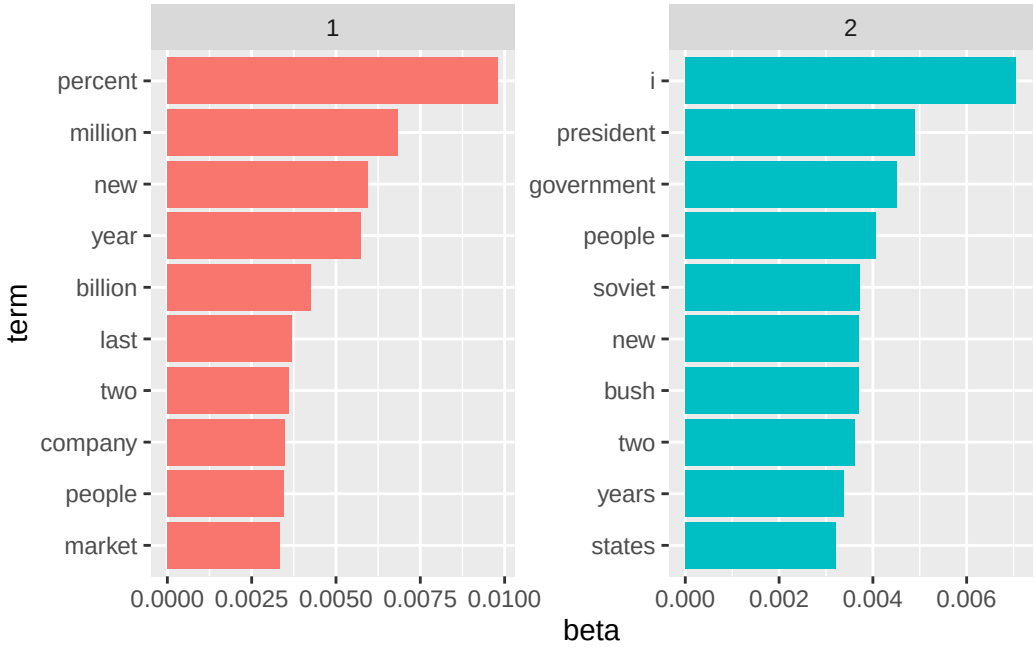
```
ap_topics <- tidy(ap_lda, matrix = "beta")
ap_topics
```

```
# A tibble: 20,946 x 3
   topic term           beta
   <int> <chr>         <dbl>
 1     1 aaron      1.69e-12
 2     2 aaron      3.90e- 5
 3     1 abandon    2.65e- 5
 4     2 abandon    3.99e- 5
 5     1 abandoned  1.39e- 4
 6     2 abandoned  5.88e- 5
 7     1 abandoning 2.45e-33
 8     2 abandoning 2.34e- 5
 9     1 abbott     2.13e- 6
10     2 abbott     2.97e- 5
# i 20,936 more rows
```

Notice that this has turned the model into a one-topic-per-term-per-row format. For each combination, the model computes the probability of that term being generated from that topic.

We could use dplyr's `slice_max()` to find the 10 terms that are most common within each topic. As a tidy data frame, this lends itself well to a ggplot2 visualization.

```
ap_top_terms <- ap_topics %>%
  group_by(topic) %>%
  slice_max(beta, n = 10) %>%
  ungroup() %>%
  arrange(topic, -beta)

ap_top_terms %>%
  mutate(term = reorder_within(term, beta, topic)) %>%
  ggplot(aes(beta, term, fill = factor(topic))) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~ topic, scales = "free") +
  scale_y_reordered()
```



This visualization lets us understand the two topics that were extracted from the articles. The most common words in topic 1 include "percent", "million", "billion", and "company", which suggests it may represent business or financial news. Those most common in topic 2 include "president", "government", and "soviet", suggesting that this topic represents political

news. One important observation about the words in each topic is that some words, such as "new" and "people", are common within both topics. This is an advantage of topic modeling as opposed to "hard clustering" methods: topics used in natural language could have some overlap in terms of words.

As an alternative, we could consider the terms that had the greatest difference in β between topic 1 and topic 2. This can be estimated based on the log ratio of the two: $\log_2\left(\frac{\beta_2}{\beta_1}\right)$ (a log ratio is useful because it makes the difference symmetrical: β_2 being twice as large leads to a log ratio of 1, while β_1 being twice as large results in -1). To constrain it to a set of especially relevant words, we can filter for relatively common words, such as those that have a β greater than 1/1000 in at least one topic.

```
beta_wide <- ap_topics %>%
  mutate(topic = paste0("topic", topic)) %>%
  pivot_wider(names_from = topic, values_from = beta) %>%
  filter(topic1 > .001 | topic2 > .001) %>%
  mutate(log_ratio = log2(topic2 / topic1))

beta_wide
```

```
# A tibble: 198 x 4
   term              topic1      topic2 log_ratio
   <chr>              <dbl>       <dbl>     <dbl>
 1 administration 0.000431  0.00138         1.68
 2 ago            0.00107   0.000842       -0.339
 3 agreement      0.000671  0.00104         0.630
 4 aid            0.0000476 0.00105         4.46
 5 air            0.00214   0.000297       -2.85
 6 american       0.00203   0.00168        -0.270
 7 analysts       0.00109   0.000000578   -10.9
 8 area           0.00137   0.000231       -2.57
 9 army           0.000262  0.00105         2.00
10 asked          0.000189  0.00156         3.05
# i 188 more rows
```

The words with the greatest differences between the two topics are visualized in the figure.

We can see that the words more common in topic 2 include political parties such as "democratic" and "republican", as well as politician's names such as "dukakis" and "gorbachev". Topic 1 was more characterized by currencies like "yen" and "dollar", as well as financial terms such as "index", "prices" and "rates". This helps confirm that the two topics the algorithm identified were political and financial news.

Document-topic probabilities

Besides estimating each topic as a mixture of words, LDA also models each document as a mixture of topics. We can examine the per-document-per-topic probabilities, called γ ("gamma"), with the matrix = "gamma" argument to `tidy()`.

```
ap_documents <- tidy(ap_lda, matrix = "gamma")
ap_documents
```

```
# A tibble: 4,492 x 3
   document topic    gamma
      <int> <int>    <dbl>
 1        1     1 0.248
 2        2     1 0.362
 3        3     1 0.527
 4        4     1 0.357
 5        5     1 0.181
 6        6     1 0.000588
 7        7     1 0.773
 8        8     1 0.00445
 9        9     1 0.967
10       10     1 0.147
# i 4,482 more rows
```

Each of these values is an estimated proportion of words from that document that are generated from that topic. For example, the model estimates that only about 25% of the words in document 1 were generated from topic 1.

We can see that many of these documents were drawn from a mix of the two topics, but that document 6 was drawn almost entirely from topic 2, having a γ from topic 1 close to zero. To check this answer, we could `tidy()` the document-term matrix and check what the most common words in that document were.

```
tidy(AssociatedPress) %>%
  filter(document == 6) %>%
  arrange(desc(count))
```

```
# A tibble: 287 x 3
   document term           count
      <int> <chr>          <dbl>
 1        6 noriega           16
```

```
 2        6 panama            12
 3        6 jackson            6
 4        6 powell             6
 5        6 administration     5
 6        6 economic           5
 7        6 general            5
 8        6 i                  5
 9        6 panamanian         5
10        6 american           4
# i 277 more rows
```

Based on the most common words, this appears to be an article about the relationship between the American government and Panamanian dictator Manuel Noriega, which means the algorithm was right to place it in topic 2 (as political/national news).