Lab 7b - Text Analysis - Example Solution

Lab Purpose

We want to learn about text analysis, and will continue to focus on Emily Dickinson's poetry which we saw in our Scraping lab. The College has a connection to Emily Dickinson that you might know about. The Emily Dickinson Museum is located within walking distance of the Amherst College campus, and is overseen by Amherst.

We're going to analyze Dickinson's poetry. Recall that we scraped the text for her poems from Wikipedia (the final web scraping code used is available in the Lab 5 folder of the class repo in the `scrape-poems.R` script file, which you engaged with in the prep).

We'll learn about functions in the following packages during our lab:

1. the **tidytext** package, which makes text analysis easier and is consistent with the tools we've been using in the **tidyverse** package;

2. the **wordcloud** package which allows us to visually represent the text data in word clouds; and

3. the **textdata** package which allows us to access lexicons for sentiment analysis.

Make sure you load each package by running the `setup` code chunk above (and that you've installed them if you are working on your own machine!).

The Data

We'll be working with the set of Emily Dickinson's poems we scraped from Wikipedia, available in the file "dickinson-poems.txt" (although it's a txt file, we can load this file using `read_csv()`). We'll remove the cases where the poem text is missing.

```
# Load dataset and name it "poems" for easier reference
poems <- readRDS("DickinsonPoems.Rds")
```

Reminders: Filepaths and Working Directories

We've chatted about this a few times in class and in various emails, but a quick reminder that it's important to know where your data is in relation to your .Qmd/.Rmd and what your working directory is set to in R. This will be very important for your Shiny apps in particular - the apps have to be able to load the data without you setting the working directory every time.

When you knit a file, it searches for files *relative to where the file is located*. Here are four use cases for reading in a file called "my-data.csv" that you can generalize to other situations:

1. The dataset is in the same folder as the Qmd/Rmd file:

```
mydata <- read_csv("my-data.csv")
```

2. The dataset is in a subfolder called "data" within the folder the Qmd/Rmd file is in:

```
mydata <- read_csv("data/my-data.csv")
```

3. The dataset is up a folder relative to the folder the Qmd/Rmd file is in (e.g., the data are in a folder called "my-project" and the Rmd file is in "my-project/code"):

```
mydata <- read_csv("../my-data.csv")
```

4. The dataset is in one subfolder and the Qmd/Rmd file is in another subfolder of the same directory (e.g., the data has path "my-project/data" and the Qmd/Rmd file has path "my-project/code"):

```
mydata <- read_csv("../data/my-data.csv")
```

When you are coding interactively, you'll want to make sure your working directory is the same as the folder your Qmd/Rmd file is in so that the filepaths you specify work. You can check your working directory in the console by typing `getwd()` and hitting enter. If it doesn't match, one way to set your working directory in RStudio is to use the **Session** tab, then **Set Working Directory** and use **To Source File Location**. That's the way I've been sharing with many of you. A second way is to navigate to the folder in the **Files** pane, click on the **More** gear menu in that pane, and choose **Set As Working Directory**.

1 - Tidying text

In this part of the lab, we'll work through pre-processing a text using the **tidytext** package.

Tokenizing a text is the process of splitting the text from it's full form and splitting it into smaller units (e.g., sentences, lines, words, etc.). We do this with the `unnest_tokens()` function from the **tidytext** package, which takes on two main arguments: `output` and `input`.

`output` creates a new variable that will hold the smaller units of text, and `input` identifies the variable in your dataframe that holds the full text. In the process, we get a long version of the dataset.

> part a - Run the code below and view the `poems_words_all` dataset and compare it to the `poems` dataset to see these changes.

```
poems_words_all <- poems %>%
  unnest_tokens(output = word, input = text)

dim(poems)
```

```
[1] 1777    2
```

```
dim(poems_words_all)
```

```
[1] 86371     2
```

In the poems data set, the two variables are poem title and the poem text. In poems_words_all, the two variables are poem title and word from the poem text. The poem text is broken apart so each word has its own row. As a result, we go from 1777 poems to 86371 words.

The default unit for tokens is a word, but you can specify the `token =` option to tokenize the text by other functions, such as "characters", "ngrams" (n words that occur together), "sentences", or "lines", among other options.

> part b - Try one or more of these alternative options, using the help as a guide, and see how it changes the output (another option has been shown).

Solution:

```
poems_ngrams <- poems %>%
  unnest_tokens(output = bigram, input = text,
```

```
                token = "ngrams", n = 2)
```

Here, we create bigrams, meaning 2 words together. You could create trigrams, sentences, or lines, etc. Bigrams are fairly common due to being able to look for phrases. We get 89434 bigrams from our poems here.

Code for another option is shown below.

```
# 368633 characters!
poems_characters <- poems %>%
  unnest_tokens(output = character, input = text,
                token = "characters")
```

Stop Words

Many commonly used words like "the", "if", and "or" don't provide any insight into the text and are not useful for analysis. These are called *stop words* and are typically removed from a body of text before analysis. The **tidytext** package provides a dataframe with stop words from three different lexicons (SMART, snowball, and onix). We can use this `stop_words` dataset and `anti_join()` to remove all the stop words from our `poems_words_all` dataset.

```
data(stop_words)

# First, take a look at the stop_words dataset
head(stop_words)
```

```
# A tibble: 6 x 2
  word      lexicon
  <chr>     <chr>
1 a         SMART
2 a's       SMART
3 able      SMART
4 about     SMART
5 above     SMART
6 according SMART
```

```
tail(stop_words)
```

```
# A tibble: 6 x 2
```

```
  word     lexicon
  <chr>    <chr>
1 you      onix
2 young    onix
3 younger  onix
4 youngest onix
5 your     onix
6 yours    onix
```

```
stop_words %>%
  count(lexicon)
```

```
# A tibble: 3 x 2
  lexicon      n
  <chr>    <int>
1 SMART      571
2 onix       404
3 snowball   174
```

```
# Create new dataset since we are removing words
poems_words <- poems_words_all %>%
  anti_join(stop_words, by = "word")

# Explore which stop words were removed
## If you don't want all these words removed, you can modify
## the stop_words dataframe before `anti_join`ing
removed <- poems_words_all %>%
  anti_join(poems_words, by = "word") %>%
  count(word) %>%
  arrange(word)
```

part c - Look at the removed dataset. What do you think? Are there any words that have been removed that you think might be meaningful to keep?

Solution:

This removes many rows from our data set. The new data set is only 35105 rows. In total, we see that 551 words were removed, but they accounted for over 50k rows in the data set. This makes sense if you look at the removed data set, where we can see "a" was removed over 2000 times. However, it is possible that we don't want to remove some of these words. For example, if we were studying politeness, we might want to keep words like "please" and "thank" in the

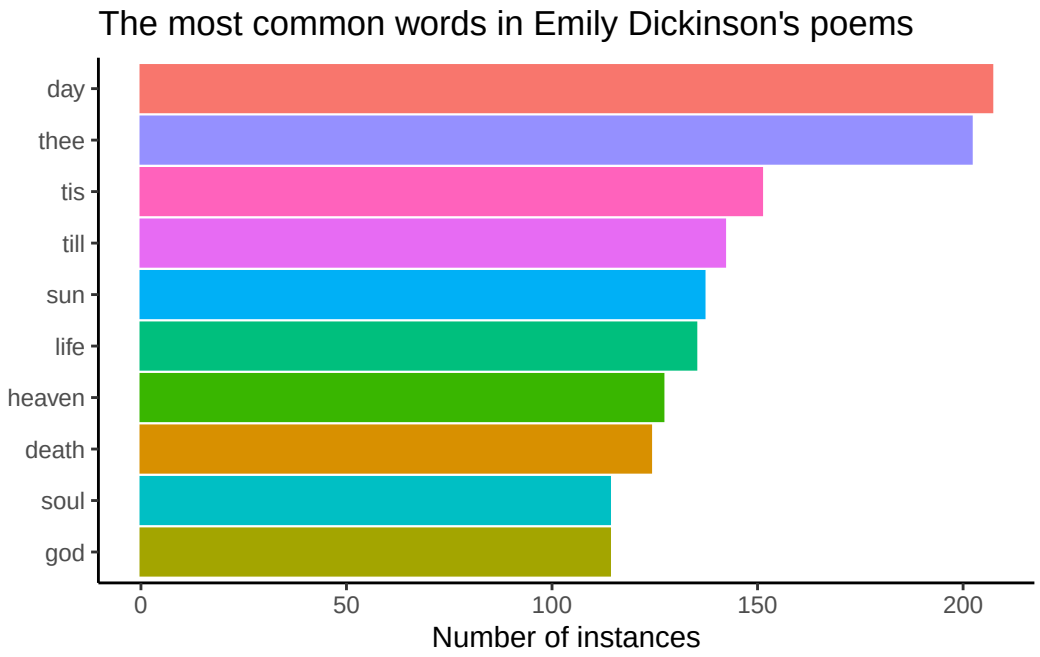
data set. If that's the case, you need to make sure you modify the stop_words list before doing the anti_join.

2- Term frequency

Once our text has been pre-processed, we can use functions we already know and love to create a simple descriptive analysis of the term frequency.

> part a - Common words plot. Run the code below to create a simple plot of the 10 most common words used by Emily Dickinson. *Note*: The `slice()` function is used to select a subset of rows from a dataframe.

```
poems_words |> #this is an alternative pipe, but it's still a pipe operator
  count(word, sort = TRUE) |>
  slice(1:10) |>
  # fct_reorder is used to re-order the axis (displaying the word)
  # by values of n (the number of times that word was used)
  ggplot(aes(x = fct_reorder(word, n), y = n, color = word, fill = word)) +
  geom_col() +
  # Rotate graph
  coord_flip() +
  guides(color = "none",
         fill = "none") +
  labs(
    # Remove x variable label; notice that although coordinates are flipped,
    # the labels correspond to which variables were specified
    # as `x` and `y` in `aes()`
    x = NULL,
    y = "Number of instances",
    title = "The most common words in Emily Dickinson's poems")
```

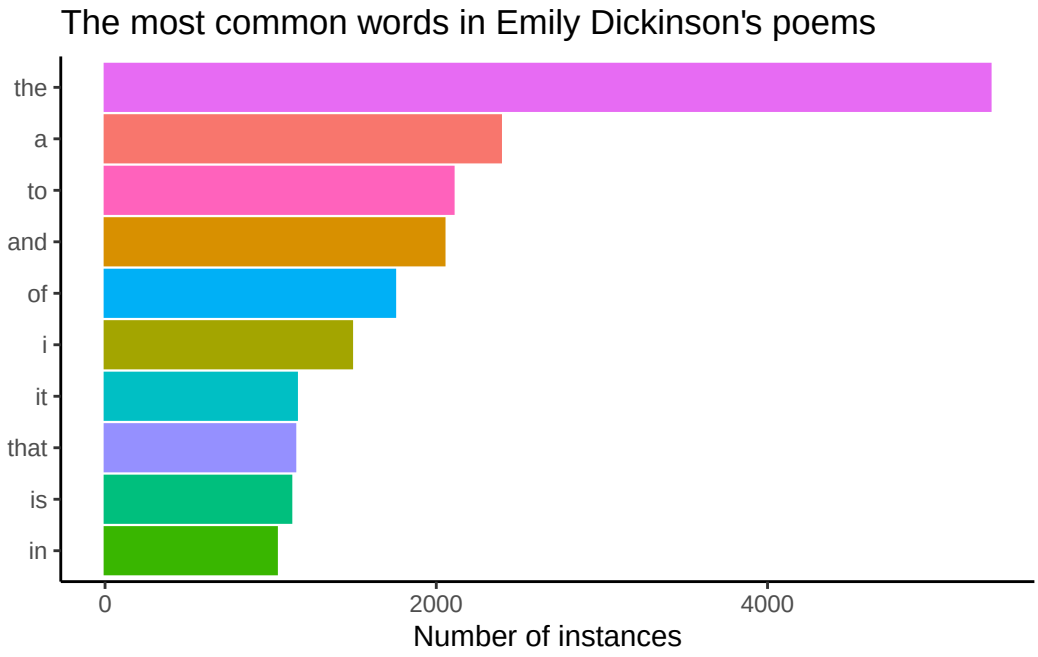


part b - Run the same code but using the `poems_words_all` dataset. What do you notice about this graphic, and what does this tell us about the utility of removing stop words before analysis?

Solution:

```
poems_words_all |>
  count(word, sort = TRUE) |>
  slice(1:10) |>
  # fct_reorder is used to re-order the axis (displaying the word)
  # by values of n (the number of times that word was used)
  ggplot(aes(x = fct_reorder(word, n), y = n, color = word, fill = word)) +
  geom_col() +
  # Rotate graph
  coord_flip() +
  guides(color = "none",
         fill = "none") +
  labs(
    # Remove x variable label; notice that although coordinates are flipped,
    # the labels correspond to which variables were specified
    # as `x` and `y` in `aes()`
    x = NULL,
    y = "Number of instances",
```

```
    title = "The most common words in Emily Dickinson's poems")
```



These 10 words are fairly useless for analyzing the text. Not surprisingly, we see that this list includes words like "the" and "a" which aren't going to help us analyze the poetry. In fact, all ten of these words are stop words! This should reinforce the understanding that removing stop words is very important.

Recap

To recap, it really only took 4 lines of code to get from our scraped dataset to a dataset formatted for plotting word frequencies:

```
word_frequencies <- poems %>%
  unnest_tokens(output = word, input = text) %>%
  anti_join(stop_words, by = "word") %>%
  count(word, sort = TRUE)
```

part c - **Your turn!** Create a simple plot of the 10 most common *bigrams* used by Emily Dickinson. *Hint: you need to remove stop words AFTER creating the dataset with one row per bigram. (Why?) Create one row per bigram using the*

`unnest_tokens` function; then, remove any row that has at least one stop word in it.

```
# one approach to removing bigrams with any stop word in it

# not everyone would have made bigrams above
poems_bigrams <- poems |>
  unnest_tokens(output = bigram, input = text, token = "ngrams", n = 2)

poems_bigrams_clean <- poems_bigrams |>
  separate(col="bigram", into=c("word1", "word2"), sep=" ", remove=FALSE) |>
  anti_join(stop_words, by=c("word1"="word")) |>
  anti_join(stop_words, by=c("word2"="word")) |>
  filter(!is.na(bigram)) |>
  count(bigram, sort = TRUE) |>
  slice(1:10)

head(poems_bigrams_clean) |> kable()
```

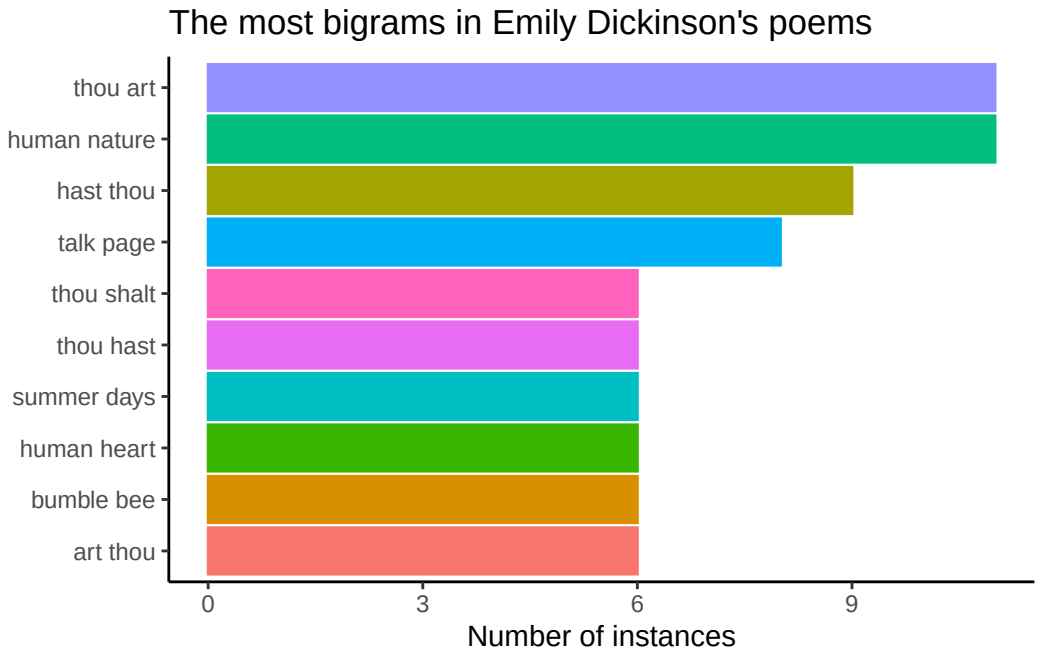
bigram	n
human nature	11
thou art	11
hast thou	9
talk page	8
art thou	6
bumble bee	6

```
# a second approach
poems_bigrams_clean <- poems_bigrams |>
  separate(col="bigram", into=c("word1", "word2"), sep=" ", remove=FALSE) |>
  filter(!(word1 %in% stop_words$word | word2 %in% stop_words$word)) |>
  filter(!is.na(bigram)) |>
  count(bigram, sort = TRUE) |>
  slice(1:10)

head(poems_bigrams_clean) |> kable()
```

bigram	n
human nature	11
thou art	11
hast thou	9
talk page	8
art thou	6
bumble bee	6

```
ggplot(poems_bigrams_clean, aes(x = fct_reorder(bigram, n), y = n,
  color = bigram, fill = bigram)) +
  geom_col() +
  # Rotate graph
  coord_flip() +
  guides(color = "none",
  fill = "none") +
  labs(
    # Remove x variable label; notice that although coordinates are flipped,
    # the labels correspond to which variables were specified
    # as `x` and `y` in `aes()`
    x = NULL,
    y = "Number of instances",
   title = "The most bigrams in Emily Dickinson's poems")
```



3 - Word clouds

Word clouds can be used as a quick visualization of the prevalence of words in a corpus.

> part a - Basic word cloud. We can get a bare-bones word cloud using the `wordcloud()` function from the **wordcloud** package. Run the code chunk below to try this out.

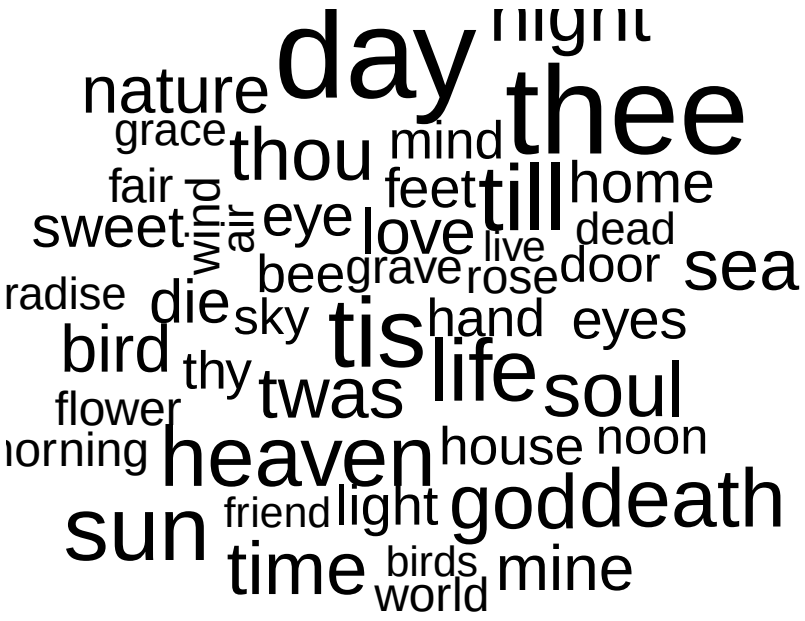
Note: if you get the error: "Error in plot.new() : figure margins too large" or a message "[word] could not be fit on page. It will not be plotted.", try re-adjusting the size of the plotting pane before re-running the code.

Syntax: We are working with packages that do not all recognize the piping operator and do not use some of our other coding conventions. Whenever you use a new package, you need to learn its syntax. Here, we see a common construction - dataset$variable - that is needed to access variables in data sets if you are working in base R. Sometimes there are alternative syntaxes that help bridge the gap (using `with` to specify the data set that the variables are in is demo-ed below).

```
# Word cloud will rearrange each time unless seed is set
# Seed is set before each call to wordcloud, generates 3 identical clouds

# Preferred syntax, closest to class coding style we can get
# Using pipes with *with*
set.seed(231)
word_frequencies %>%
  with(wordcloud(words = word, freq = n, max.words = 50))

# *With* syntax without pipes
set.seed(231)
with(word_frequencies, wordcloud(words = word, freq = n, max.words = 50))
```



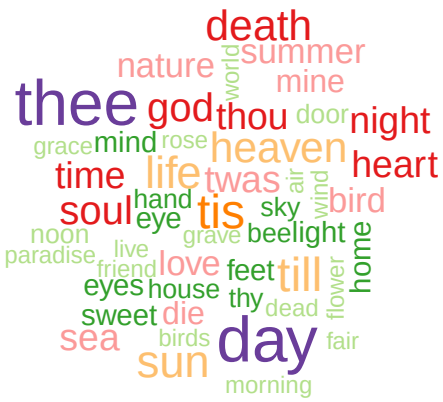

```
# Using base R to reference variables directly
set.seed(231)
wordcloud(words = word_frequencies$word,
          freq = word_frequencies$n,
          max.words = 50)
```

part b - Custom word cloud. We can customize the word cloud by mapping the size and color of words to their frequency. Try this out below.

```
# choose color palette from color brewer
mypal <- brewer.pal(10, "Paired")

set.seed(231)
word_frequencies %>%
```

```
with(wordcloud(words = word,
               freq = n,
               min.freq = 20,
               max.words = 50,
               # plot the words in a random order
               random.order = TRUE,
               # specify the range of the size of the words
               scale = c(2, 0.3),
               # specify proportion of words with 90 degree rotation
               rot.per = 0.15,
               # colors words from least to most frequent
               colors = mypal,
               # font family
               family = "sans"))
```

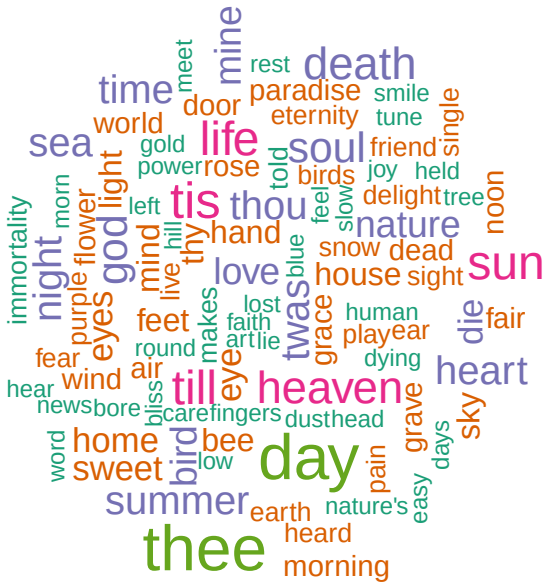


part c - Create your own custom word cloud with *100* words and change the color scheme. Don't forget to set a seed to be able to reproduce the word cloud.

Solution:

```r
# choose color palette from color brewer
mypal <- brewer.pal(5, "Dark2")

set.seed(231)
word_frequencies %>%
  with(wordcloud(words = word,
                 freq = n,
                 min.freq = 25,
                 max.words = 100,
                 # plot the words in a random order
                 random.order = TRUE,
                 # specify the range of the size of the words
                 scale = c(2, 0.5),
                 # specify proportion of words with 90 degree rotation
                 rot.per = 0.3,
                 # colors words from least to most frequent
                 colors = mypal,
                 # font family
                 family = "sans"))
```



4 - Term frequency-inverse document frequency (tf-idf)

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 corpus. That may seem slightly counter-intuitive - but you still have ways of accessing the commonly used words if that's what you want to check out instead.

Computing term frequency statistics

The `bind_tf_idf()` function will compute the term frequency (tf), inverse document frequency (idf), and the tf-idf statistics for us. It requires a dataset with one row per word per poem, meaning we need a variable to indicate which poem the word comes from (`title`, in our tokenized dataset), a variable to indicate the word (`word` in the tokenized dataset), and a third variable to indicate the number of times that word appears in that specific poem. We have to compute that count and add it to our dataset to proceed.

> part a - Run the code below to generate the tf-idf statistics for our data. Note that we do not need to remove stop words. Why not?

Solution:

```
# Get remaining missing variable
word_freqs_by_poem <- poems %>%
  unnest_tokens(output = word, input = text) %>%
  group_by(title) %>%
  # count does what summarize(n()) does, leaves new variable "n"
  count(word)

# Compute tf_idf statistics
poems_tfidf <- word_freqs_by_poem %>%
  bind_tf_idf(term = word,
              document = title,
              n = n)
```

We don't have to remove stop words because the tf-idf takes into account word frequency. For words like "a", "the" and "or" that occur very frequently, their inverse document frequency will be very low and therefore their tf-idf will be very low.

You could argue in fact that you want to leave all the stopwords in place because otherwise, the tf_idf calculations won't be run on the "entire" poem, just the text with stopwords removed.

> part b - Glimpse the tfidf data set and be sure you understand what each column represents.

Solution:

```
glimpse(poems_tfidf)
```

```
Rows: 67,234
Columns: 6
Groups: title [1,666]
$ title  <chr> "\"And with what body do they come?\" -", "\"And with what body~
$ word   <chr> "a", "and", "bethlehem", "body", "come", "do", "door", "eyes", ~
$ n      <int> 1, 2, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, ~
$ tf     <dbl> 0.02083333, 0.04166667, 0.02083333, 0.04166667, 0.04166667, 0.0~
$ idf    <dbl> 0.4559374, 0.5938072, 5.8087429, 5.2209562, 2.8748860, 2.535378~
$ tf_idf <dbl> 0.009498695, 0.024741965, 0.121015477, 0.217539844, 0.119786918~
```

Title and word carryover from the poems dataset data processing. Then we see four columns with the tf_idf statistics: n, tf, idf, and tf_idf. n is the count of the number of times that word appears in the poem. tf is the frequency of the word in that poem (n/total number words in the poem). idf is a bit more complicated (formula in text), but it measures the prevalence of the word across the set of poems. Finally, tf_idf is the product of tf and idf. Poems with a high tf_idf value for a particular word contain that word MANY times more than other poems, speaking relatively.

part c - Visualizing tf-idf. We can visualize the words with the highest 10 tf-idf values for a subset of the poems using the code below. *Change the seed* to get your own 4 randomly selected poems.

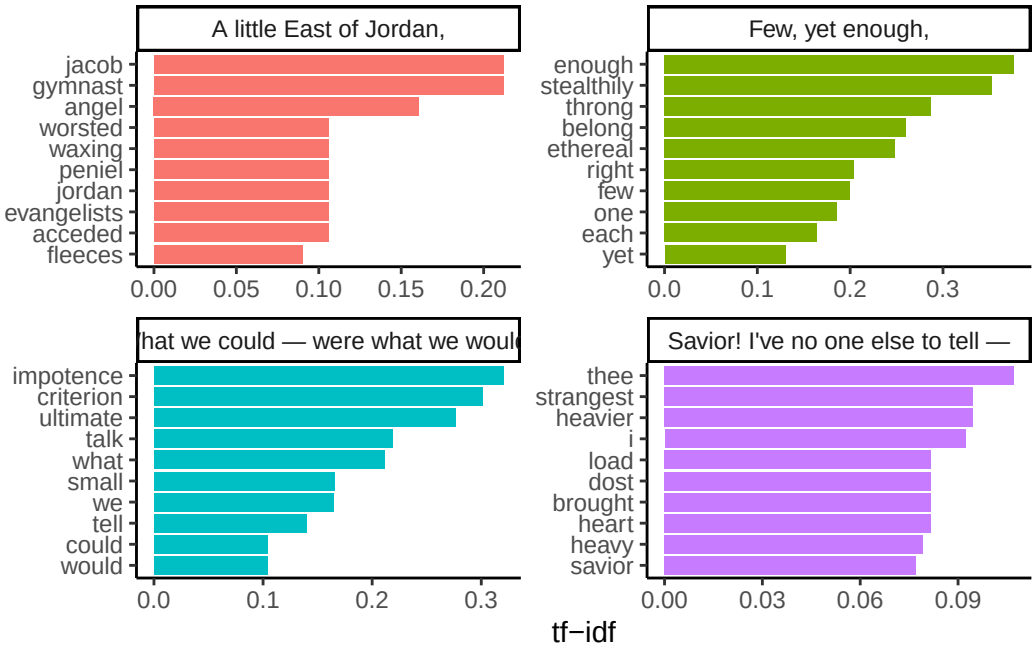
Solution:

```
set.seed(231) #change this!
poems_subset <- sample(poems$title, size = 4)

# Compute tf-idf for each of the 4 poems
top_tfidf <- poems_tfidf %>%
  filter(title %in% poems_subset) %>%
  arrange(desc(tf_idf)) %>%
  group_by(title) %>%
  slice(1:10) %>%
  ungroup()

# Plot top 10 tf-idf words
ggplot(data = top_tfidf, aes(x = reorder(word, tf_idf), y = tf_idf,
                             fill = title)) +
```

```
geom_col(show.legend = FALSE) +
facet_wrap(~ title, ncol = 2, scales = "free") +
coord_flip() +
labs(x = NULL, y = "tf-idf")
```



5 - Sentiment analysis

What is sentiment analysis?

From Text Mining with R (Silge & Robinson 2019):

"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… 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 sentiment analysis, but it is an often-used approach."

There are different lexicons that can be used to classify the sentiment of text. Today, we'll compare two different lexicons that are both based on unigrams, the AFINN lexicon and the NRC lexicon.

The AFINN lexicon (Nielsen 2011) assigns words a score from -5 (negative sentiment) to +5 (positive sentiment).

> part a - Check out the AFINN lexicon using the code below. What do you think of the scores? What is the rating for the word "slick"? Does "slick" always have a positive connotation (can you think of a sentence where "slick" has a negative connotation)?

Solution:

```
# Type "Yes" to download if prompted
afinn_lexicon <- get_sentiments("afinn")

mosaic::favstats(~ value, data = afinn_lexicon)
```

```
 min Q1 median Q3 max       mean       sd    n missing
  -5 -2     -2  2   5 -0.5894227 2.123931 2477       0
```

```
afinn_lexicon %>%
  filter(word == "slick")
```

```
# A tibble: 1 x 2
  word  value
  <chr> <dbl>
1 slick     2
```

There are only 5 words that get assigned a value of 5 (most positive) and 16 words that get assigned a value of -5 (most negative). The majority of words are assigned values between -2 and 2, which probably reflects the fact that their sentiment can change depending on context, rather than that they're truly that neutral in every context.

Slick has rating +2. This is interesting because "slick" is sometimes used to refer to somebody who is trying to get away with something.

> part b - Use the `get_sentiments()` function to create a dataframe `nrc_lexicon` that holds the NRC Word-Emotion Association lexicon (Mohammad 2010). What does each row in this dataset represent?

Hint: it's *not* the same as the `afinn_lexicon` dataset.

Solution:

```
nrc_lexicon <- get_sentiments("nrc")

head(nrc_lexicon)
```

```
# A tibble: 6 x 2
  word      sentiment
  <chr>     <chr>
1 abacus    trust
2 abandon   fear
3 abandon   negative
4 abandon   sadness
5 abandoned anger
6 abandoned fear
```

Each row in the data set is a word-sentiment pair. Words may have multiple rows if they could represent multiple sentiments. This is very different from the afinn rating system.

The NRC lexicon categorizes words as yes/no for the following sentiment categories: positive, negative, anger, anticipation, disgust, fear, joy, sadness, surprise, and trust.

> part c - User (and Consumer!) Beware: Do you see any issues in applying these lexicons (developed fairly recently) to the Emily Dickinson poems?

Solution:

They may not be appropriate given how language evolves over time. The meaning and sentiment of a particular word may have changed over the years, such that the poet's intended sentiment is not how it would be classified today. In addition, many words may not be included in the lexicons because of the differences in language used in the 1800s and in the 2000s.

(Note that many words used today are also not included in some of the lexicons because they are neutral, having neither a particularly positive or negative sentiment).

> part d - The lexicons are based on unigrams. Do you see any disadvantages of basing the sentiment on single words?

Solution:

It does not account for qualifiers (e.g. "not good" only counts "good"; "somewhat happy" only counts "happy"). A text with many qualifiers could be rated incorrectly as being mostly positive or negative because it does not consider the negating.

> part e - We can calculate how many words used in the poems are not found in the lexicons using the code below. List a few words that are not in the NRC lexicon that appear in the poems. What proportion of unigrams observed within this corpora of Dickinson poems are *not* scored by the NRC lexicon?

Solution:

```
# assumes you called the nrc lexicon nrc_lexicon, as asked above
nrc_missed_words <- word_frequencies %>%
  anti_join(nrc_lexicon, by = "word")

nrow(nrc_missed_words)/nrow(word_frequencies)
```

```
[1] 0.7798844
```

A little under 78% of words in Dickinson's poems are not scored by the NRC lexicon. Missed words include words like "soul", "heaven" and "night", which was surprising to me.

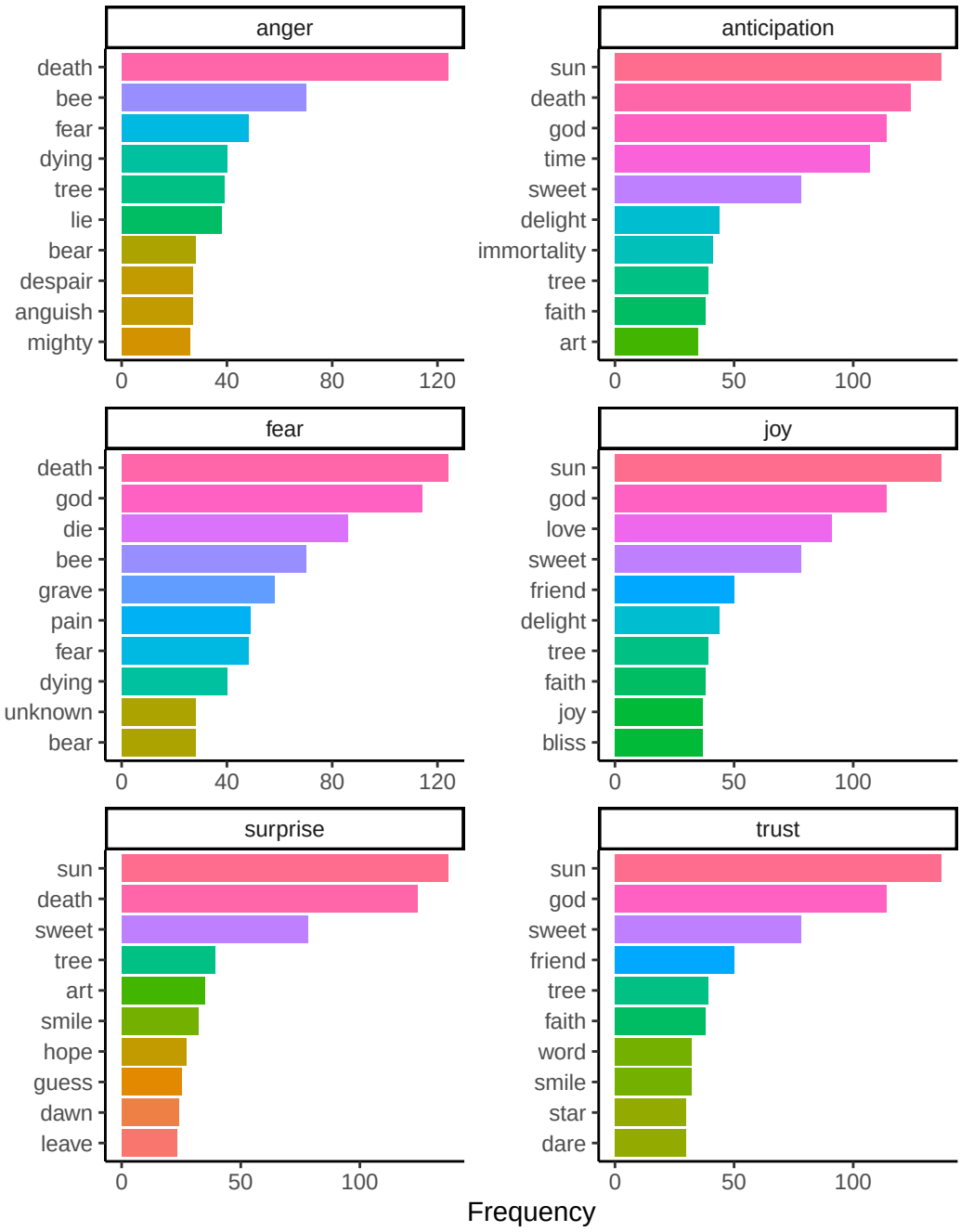
> part f - With these (rather important!) drawbacks in mind, let's go ahead and view the top words by sentiment classified by the NRC lexicon. That is, create a figure of the top 10 words under each sentiment, facetted by sentiment, for the following sentiments: anger, anticipation, fear, joy, surprise, and trust. Use code given in earlier chunks to guide you.

Solution:

```
nrc_poems <- word_frequencies %>%
  inner_join(nrc_lexicon, by = "word") %>%
  filter(sentiment %in% c("anger", "anticipation",
                          "fear", "joy",
                          "surprise", "trust")) %>%
  arrange(sentiment, desc(n)) %>%
```

```
  group_by(sentiment) %>%
  slice(1:10)

ggplot(data = nrc_poems, aes(x = reorder(word, n), y = n,
                             fill = as.factor(n))) +
  geom_col(show.legend = FALSE) +
  coord_flip() +
  facet_wrap(~ sentiment, ncol = 2, scales = "free") +
  labs(x = NULL, y = "Frequency")
```



part g - How might you summarize the sentiment of this corpus using the AFINN lexicon?

Solution:

Based on the AFINN lexicon, the collection of Dickinson's poems are positive on the whole, with a total score of 1111, indicating more positive-valued words than negative-valued words overall.

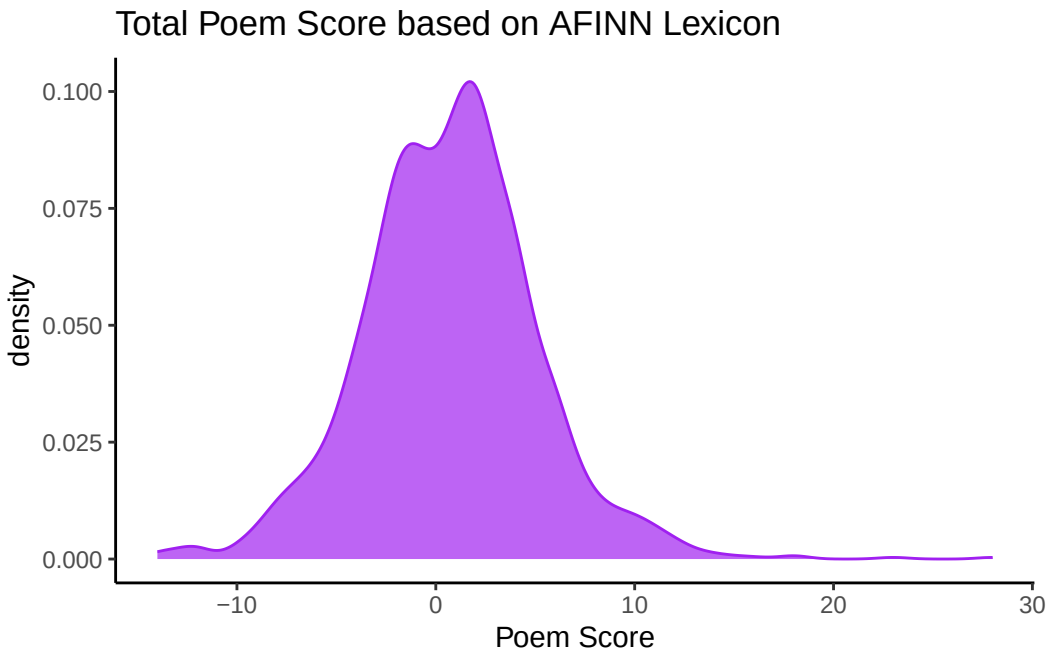
Alternatively, one could compute a total score for each poem, and then consider the distribution of scores across poems. In this case, across the 1,772 poems, half actually have a total score less than 1 (so would lean neutral to negative). About 25% have a total score of 3 or more so lean positive. It seems like some particularly positive poems are driving the overall score to lean positive. The density plot shows us a bimodal distribution, with many poems scoring either slightly negative or slightly positive.

```
# Overall score
poems_words_all %>%
  inner_join(afinn_lexicon, by = "word") %>%
  summarize(total_score = sum(value))
```

```
  total_score
1        1059
```

```
# Scores by poem
afinn_by_poem <- poems_words_all %>%
  inner_join(afinn_lexicon, by = "word") %>%
  group_by(title) %>%
  summarize(poem_score = sum(value)) %>%
  # arrange useful for viewing data set, not necessary to proceed with plot
  arrange(poem_score)

ggplot(data = afinn_by_poem, aes(x = poem_score)) +
  geom_density(color = "purple",
               fill = "purple",
               alpha = 0.7) +
  labs(title = "Total Poem Score based on AFINN Lexicon",
       x = "Poem Score")
```




```
mosaic::favstats(data = afinn_by_poem, ~ poem_score)
```

```
 min Q1 median Q3 max      mean       sd    n missing
 -14 -2      1  3  28 0.7102616 4.392247 1491       0
```

6 - Regular Expressions

When working with text as data, regular expressions are a powerful tool to find patterns. They have enormous flexibility, but often provide challenges to learners. Do I need a slash? Am I detecting the pattern I'm looking for?

Here, we explore applying some regular expressions to the titles of Emily Dickinson poems. You should think about what would change to apply these to individual poem text.

First, we get just the titles in their own vector (note, NOT a data frame).

```
titles <- poems$title
```

What poems had the word Bee with a space - "Bee" or "bee" in their title?

```
titles %>% str_subset("(B|b)ee ") %>% head()
```

```
[1] "A Bee his burnished Carriage"
[2] "Because the Bee may blameless hum"
[3] "I stole them from a Bee -"
[4] "Least Bee that brew -"
[5] "The Bee is not afraid of me."
[6] "The Bird did prance - the Bee did play -"
```

part a - Copy and edit the code above to look for "Bee" only. What do you find?

Solution:

```
titles %>% str_subset("Bee ") %>% head()
```

```
[1] "A Bee his burnished Carriage"
[2] "Because the Bee may blameless hum"
[3] "I stole them from a Bee -"
[4] "Least Bee that brew -"
[5] "The Bee is not afraid of me."
[6] "The Bird did prance - the Bee did play -"
```

The first 6 entries are still the same. She capitalized "Bee" in her titles it seems. In fact, it appears there are only these 6 entries. So "bee" never appeared in a poem title. The space there is important!

part b - What if the space was left out of the original expression? What else does this pick up?

Solution:

```
#Edit as needed
titles %>% str_subset("(B|b)ee") %>% head()
```

```
[1] "A Bee his burnished Carriage"
[2] "After all Birds have been investigated and laid aside -"
[3] "Because the Bee may blameless hum"
[4] "Bee! I'm expecting you!"
[5] "Bees are Black, with Gilt Surcingles -"
[6] "Fame is a bee."
```

We learn several things here. Some poems have "Bee" in the title but without the space. We also see "bee." However, we've also picked up "been".

So, if we really want to look for "Bee" or "bee", we need to be more creative. There doesn't have to be a space afterwards, but we need to be sure it's not a letter - we don't want to pick up "been".

Here is how we could get that:

```
titles %>% str_subset("(B|b)ee[^A-Za-z]")
```

```
 [1] "A Bee his burnished Carriage"
 [2] "Because the Bee may blameless hum"
 [3] "Bee! I'm expecting you!"
 [4] "Fame is a bee."
 [5] "I stole them from a Bee -"
 [6] "Least Bee that brew -"
 [7] "Partake as doth the Bee,"
 [8] "The Bee is not afraid of me."
 [9] "The Bird did prance - the Bee did play -"
[10] "The Flower must not blame the Bee -"
[11] "To make a prairie it takes a clover and one bee,"
[12] "We - Bee and I - live by the quaffing -"
```

part c - What does the following regular expression pick up?

Solution:

```
titles %>% str_subset(" [A-Z][a-z][a-z][a-z][a-z][a-z] ") %>% head()
```

```
[1] "A Coffin - is a small Domain,"
[2] "A Counterfeit - a Plated Person -"
[3] "A Dimple in the Tomb"
[4] "A first Mute Coming -"
[5] "A Flower will not trouble her, it has so small a Foot,"
[6] "A House upon the Height -"
```

This picks up 6 letter words that are capitalized and which have a space before and after them in the title.

part d - What about the following regular expression?

Solution:

```
titles %>% str_subset("^The") %>% head()
```

```
[1] "The Admirations - and Contempts - of time -"
[2] "The Angle of a Landscape -"
[3] "The Auctioneer of Parting"
[4] "The Bat is dun, with wrinkled Wings -"
[5] "The Battle fought between the Soul"
[6] "The Bee is not afraid of me."
```

This gives all titles that start with "The".

part e - And finally, what about the following regular expression?

Solution:

```
titles %>% str_subset("ing$") %>% head()
```

```
[1] "A Light exists in Spring"
[2] "A little Madness in the Spring"
[3] "A Pang is more conspicuous in Spring"
[4] "All forgot for recollecting"
[5] "Angels, in the early morning"
[6] "Because that you are going"
```

This shows all poems ending with a word ending in "ing". No punctuation is allowed at the end.

part f - Write your own regular expression to look for a pattern of your choosing in the poem titles. Be sure it works as expected.

Solution:

```
# Answers will vary
```

References

AFINN Lexicon

Nielsen, FA. A new ANEW: Evaluation of a word list for sentiment analysis in microblogs. Proceedings of the ESWC2011 Workshop on 'Making Sense of Microposts': Big things come in small packages 718 in CEUR Workshop Proceedings 93-98. 2011 May. http://arxiv.org/abs/1103.2903.

NRC Lexicon

Mohammad S, Turney P. Crowdsourcing a Word-Emotion Association Lexicon. *Computational Intelligence*. 2013;29(3):436-465.

Mohammad S, Turney P. Emotions Evoked by Common Words and Phrases: Using Mechanical Turk to Create an Emotion Lexicon. In Proceedings of the NAACL-HLT 2010 Workshop on Computational Approaches to Analysis and Generation of Emotion in Text, June 2010, LA, California. http://saifmohammad.com/WebPages/NRC-Emotion-Lexicon.htm

Text Mining with R

Silge J, Robinson D (2016). "tidytext: Text Mining and Analysis Using Tidy Data Principles in R." *JOSS*, *1*(3). doi: 10.21105/joss.00037.

Silge J, Robinson D (2017). Text Mining with R: A Tidy Approach. O'Reilly Media Inc. Sebastopol, CA. https://www.tidytextmining.com/index.html