---
title: 'Worksheet 6: Text Analysis'
author: "Lane Riggs"
date: "April 2, 2023"
output: pdf_document

This is the sixth in a series of worksheets for History 8510 at Clemson University. The goal of these worksheets is simple: practice, practice, practice. The worksheet introduces concepts and techniques and includes prompts for you to practice in this interactive document. When you are finished, you should change the author name (above), knit your document, and upload it to canvas. Don't forget to commit your changes as you go and push to github when you finish the worksheet.

Text analysis is an umbrella for a number of different methodologies. Generally speaking, it involves taking a set (or corpus) of textual sources, turning them into data that a computer can understand, and then running calculations and algorithms using that data. Typically, at its most basic level, that involves the counting of words.

Text analysis can be broken down into 4 general steps:

  1. Acquiring a corpus
  2. Preparing the text or Pre-processing
  3. Choosing an analytical tool
    * (There are many different tools or methods for text analysis. Take a minute and Google each of these methodologies: tf-idf, topic modeling, sentiment analysis, word vector analysis, n-grams)
  4. Analyzing the results

In this worksheet we are focusing on basic text analysis. We'll learn how to load textual data into R, how to prepare it, and then how to analyze it using tf-idf or term-frequency according to inverse document frequency.

Before doing too much, lets load a few relevant libraries. The last few you will likely need to install.
```{r message=FALSE, warning=FALSE}
library(tidyverse)
library(tidytext)
library(readtext)
library(widyr)
library(SnowballC)
```

Acquiring a Corpus

First, lets install the State of the Union package. This package contains text of all the state of the Union addresses from Washington to Trump. Run `install.packages` to install the `sotu` package.
```{r}
library(sotu)
```

This package includes both the metadata about these speeches in `sotu_meta` and the texts themselves in `sotu_texts`. Lets first look at the metadata associated with this package.

```{r}
meta <- as.data.frame(sotu_meta)
head(meta)
```

This package also includes a function that will let us write all of the files to disk. This is crucial but also an unusual step because when conducting text analysis in the real world, you will not have an R package filled with the data. Rather you will have to organize the metadata and load the files yourself. Writing these to the disk allows us to practice that step.

```{r}
file_paths <- sotu_dir(dir = "sotu_files")
head(file_paths)
```

What this does is create a new directory (sotu_files) and adds each State of the Union address as a text file. Notice each speech is its own .txt file that is comprised of just the text of the speech.

(@) Take a look at the directory in your files pane and open one of the documents.

Now lets load all these texts into R using the `readtext()` function. First look up the documentation for this function and read about it.
```{r}
sotu_texts <- readtext(file_paths)
```

Take a look at sotu_texts now. Notice that we have two columns, one filled with the text, and one with a document id.
```{r}
head(sotu_texts, n = 5)
```

Now our textual data is loaded into R but the textual data and the metadata are in two different data frames. Lets combine them. Note that this isn't the way I would typically recommend doing this but its a quirk of the SOTU data. Typically when I create a metadata spreadsheet for a textual dataset I have a column for the file name which makes joining the textual data and metadata together easier. Here, we'll need to sort the dataset so that is alphabetical and then join the two together.

```{r}
sotu_whole <- 
  sotu_meta %>%
  arrange(president) %>% # sort metadata
  bind_cols(sotu_texts) %>% # combine with texts
  as_tibble() # convert to tibble for better screen viewing
glimpse(sotu_whole)
```

Now our data is loaded into R and its ready to be pre-processed.

Pre-Processing

Tokenizing

One of the most basic pre-processing techniques for textual data is to tokenize it.

Tokenization is essentially splitting a phrase, sentence, paragraph, or an entire text document into smaller units, such as individual words or terms. Each of these smaller units are called tokens. The tokens could be words, numbers or punctuation marks but, for historians, its common to remove the numbers and punctuation too. To do this we'll create a data frame where each row contains a single word with its metadata as unit of observation.

`tidytext` provides a function called `unnest_tokens().` We can use this to convert our sotu_whole data frame into one that is tokenized. It takes three arguments:

* a tibble or data frame which contains the text
* the name of the newly created column that will contain the tokens
* the name of the column within the data frame which contains the text to be tokenized

```{r}
tidy_sotu <- sotu_whole %>%
  unnest_tokens(word, text)
tidy_sotu
```

`unnest_tokens()` also did something else that is really important: it made everything lowercase and took out all punctuation. The function contains options if we wanted to keep those elements, but for our purposes we don't.

The function `unnest_tokens()` also has an option called token. Tokenizing by word is the default but you could also tokenize by characters, ngrams, lines, or sentences.

(@)Use the documentation to tokenize the dataset into sentences:
```{r}

sotu.sentences <- sotu_whole %>%
  unnest_tokens(sentences, text, token = "sentences")

sotu.sentences

```

We've talked about n-grams loosely in class. But lets define it more formally. An n-gram is a contiguous sequence of n items from a given sample of text or speech. The n stands for the number of items. So for example, a bi-gram is sets of two words.

For example, if I had the string: "Nothing to fear but fear itself" A bi-gram would look like this:
  Nothing to, to fear, fear but, but fear, fear itself.

A tri-gram would look like this:
  Nothing to fear, to fear but, but fear itself

We can use unnest_tokens() to create n-grams for us. To do that we just have to add an extra option that defines n.
```{r}
sotu_bigrams <- sotu_whole %>%
  unnest_tokens(bigram, text, token = "ngrams", n = 2)
head(sotu_bigrams$bigram)
```

(@) Use `unest_tokens()` to create tri-grams.

```{r}

sotu.trigrams <- sotu_whole %>%
  unnest_tokens(trigram, text, token = "ngrams", n = 3)

head(sotu.trigrams$trigram)

```

Stopwords

Another crucial component of text analysis is removing stopwords. Stopwords are words like "I, he, she, of, the" that are common and don't convey meaning. Because they are highly common they don't tell us anything about the content of the text itself.

There are stopwords that come with the `tidytext` package.
```{r}
stop_words
```
This is just one example of stopwords. You can find other lists such as stopwords in other languages or [stopwords designed specifically for the 19th century.](https://www.matthewjockers.net/macroanalysisbook/expanded-stopwords-list/) Its also possible you may want to edit the list of stopwords to include some of your own. For example, if we wanted to add the word, "America" to the stopwords list we could use add_row to do so:
```{r}
stop_words_custom <- stop_words %>% add_row(word="America", lexicon="NA")
```


For now lets just remove the default stopwords. The easiest way to do that here is to do an anti-join. We join and return all rows from our table of tokens tidy_sotu where there are no matching values in our list of stopwords.

```{r}
tidy_sotu_words <- tidy_sotu %>%
  anti_join(stop_words)

tidy_sotu_words
#another way to do this would be to filter by words NOT in the stop word list like this:
filter(!word %in% stop_words$word)
```

Stemming

The third common kind of pre-process is called word stemming. This process reduces a word to its root stem. So for example: fishing becomes fish, fished becomes fish, fishes becomes fish. You can easily see how this might be useful for capturing all forms of a word.

`tidytext` doesn't have its own word stemming function. Instead we have to rely on the functions provided by `hunspell` or `SnowballC`. I prefer `SnowballC`. You may need to install it before running the below code.

```{r}
library(SnowballC)
tidy_sotu_words %>%
```

```
        mutate(word_stem = wordStem(word))
```

Now if you compare the word and word_stem columns you can see the effect that wordStem had. Notice that it works well in cases like

  citizens = citizen

But it does some odd things to words like representatives. Whether this is useful for you will depend on the question your asking (and the OCR accuracy) but its a useful technique to be familiar with nevertheless.

Analysis

Lets reset our work space and ensure that our df is loaded with single tokenized words and filter by our stopword list. Go ahead and clear your environment using the broom button and then run the below code. This code is simply everything we've run up to this point.

```{r}
meta <- as.data.frame(sotu_meta)
file_paths <- sotu_dir(dir = "sotu_files")
sotu_texts <- readtext(file_paths)
sotu_whole <-
  sotu_meta %>%
  arrange(president) %>% # sort metadata
  bind_cols(sotu_texts) %>% # combine with texts
  as_tibble()
tidy_sotu <- sotu_whole %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)
```
(@) Before we move forward, take a minute a describe the chunk of code you just ran. What does each section do and how does it reflect the workflow for a topic modeling project? What are the important steps that are unique to topic modeling?

> The code first built a corpus, with the meta and file paths variables. Then you/we had to organize the metadata and load the files. But in this case, the sotu package already contains information about each State, which is titled "sotu_meta." In the first line, we are converting sotu_meta into a dataframe called 'meta.'

>In the second line/dataframe, we are adding each State of the Union address as an individual .txt file. Then, we use the readtext function to load the addresses into a dataframe that is called sotu_texts. There's two columsn in this dataframe: one for the text, and one for the document ID. So, we then combine the dataframes.

>Next comes tokenization, or when we split a phrase or paragraph or a full text into small units. So, we tokenize the dataframe using the unnest_tokens function. Lastly, we can remove stop words like "I, the, of" etc. that may not convey meaning within the context of the address. We just want to remove default stopwords like those listed.

>anti_join is the last step and this functions joins and returns rows from tidy_sotu that have no matching values in our stopwords list.

The most basic kind of analysis we might be interested in doing is counting words. We can do that easily using the `count()` function:
```{r}
tidy_sotu %>%
```

```
  count(word, sort = TRUE)
```

Now we know that the most used word in state of the union speeches is government. But what if we wanted to look at when presidents use the words war versus the word peace?
```{r}
tidy_sotu %>%
  filter(word %in% c("war", "peace")) %>%
  count(year, word)
```

This data frame is to big to understand quickly without visualizing it. We can create a bar chart to better understand it:
```{r}
library(ggplot2)
tidy_sotu %>%
  filter(word %in% c("war", "peace")) %>%
  count(year, word) %>%
  ggplot(aes(year, n, fill = word)) +
    geom_col(position = "fill")
```

We also might want to ask about the average length of each president's state of the union address. Who had the longest speech and who had the shortest?
```{r}
tidy_sotu %>%
  count(president, doc_id)  %>%
  group_by(president) %>%
  summarize(avg_words = mean(n)) %>%
  arrange(desc(avg_words))

#when we run this code, we can see that William Taft had the longest union address, and
John Adams had the shortest address.
```

(@) Think back to the metadata that we loaded about these speeches. Why are more modern president's state of the union addresses shorter?

>It could be because more laws are in place now than there were when George Washington was president, or it could be most of the speeches are part of the two-party system and the earliest speeches were not categorized as such.

(@) Filter the dataset to address this discrepancy and then recreate these statistics:
```{r}

tidy_sotu %>%
  count(president, sotu_type, doc_id) %>%
  group_by(president,sotu_type) %>%
  summarize(avg_words = mean(n)) %>%
  arrange(desc(avg_words))

```

>A lot of these results are categorized as written rather than speeches. However, I know from giving my own speeches, that I typically write up something to go along with it.

Term Frequency

Often, the raw frequency of a term is not as useful as relative frequency. In other words, how often that word appears relative to the total number of words in a text. This ratio is called **term frequency**.

You can calculate the term frequency by dividing the total occurrences of a word by the total number of words. Typically you want to do this per document.

Here's an easy way to calculate it:
```{r}
tidy_sotu_rel.freq <- tidy_sotu %>%
  count(doc_id, word, sort = T)  %>%# count occurrence of word and sort descending
  group_by(doc_id) %>%
  mutate(n_tot = sum(n), # count total number of words per doc
         term_freq = n/n_tot)
```

We can assume that words with a high frequency in the text are more important or significant. Here we can find the words with the most significance for each president:
```{r}
tidy_sotu %>%
  count(president, word)  %>%  # count n for each word
  group_by(president) %>%
  mutate(n_tot = sum(n), # count total number of words per doc
         term_freq = n/n_tot) %>%
  arrange(desc(term_freq)) %>% # sort by term frequency
  top_n(1) %>%  # take the top for each president
  print(n = Inf) # print all rows
```
(@) The code above is commented to help you follow it. Walk through the code above, and explain what each line does in your own words. If its a function you are unfamiliar with, look up the documentation.

> We first count() to see how many times unique words were used in presidential speeches. Then, we group that data by president. Mutate() is used to see the frequency of these unique words by looking at the total occurrences of all words in each document. So we see the total number of words (n_tot) and the frequency of the words (term_freq).

>Next, we arrange() to sort the data. Top_n() selects the top row for each president in the dataset, so we can see the word that that president uses the most. Lastly, we print all the rows selected.

TF-IDF

The above measures the frequency of terms within individual documents. But what if we know about words that seem more important based on the contents of the **entire** corpus? That is where tf-idf or term-frequency according to inverse document frequency comes in.

Tf-idf measures how important a word is within a corpus by scaling term frequency per document according to the inverse of the term’s document frequency (number of documents within the corpus in which the term appears divided by the number of documents). The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

Don't worry too much about how tf-idf is calculated. But if you feel like you are a bit lost and want to understand the specifics - I recommend reading the [tf-idf wikipedia page](https://en.wikipedia.org/wiki/Tf%E2%80%93idf) and this blog post from [_Learn Data

Science_](https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/).

We'll calculate tf-idf in the next code chunk but lets talk for a second about what that number will represent. It will be:

* lower for words that appear frequently in many documents of the corpus, and lowest when the word occurs in virtually all documents.
* higher for words that appear frequently in just a few documents of the corpus, this lending high discriminatory power to those few documents.

Luckily, `tidytext` provides a function for calculating tf-idf. To calculate tf-idf the function needs a list of every word in every document and the count. Like this:
```{r}
tidy_sotu %>%
  count(doc_id, word, sort = TRUE)
```
We can feed that to the function and get the tf-idf:
```{r}
sotu.tf.idf <- tidy_sotu %>%
  count(doc_id, word, sort = TRUE)  %>%
  bind_tf_idf(word, doc_id, n)
head(sotu.tf.idf)
```

The resulting data frame has 3 columns: term frequency (tf), inverse document frequency (idf) and Tf-idf (tf_idf).

Lets look at what the words with the highest tf-idf score are.
```{r}
sotu.tf.idf %>% arrange(desc(tf_idf))
```

(@) Pick a president who served more than one term. Filter the dataset and generate both raw word counts and tf-idf scores for that president. What words are most significant in each method? Why and what does that tell you about that president?

```{r}

tidy_sotu %>%
  filter(president == "Barack Obama") %>%
  count(doc_id, word, sort = TRUE)

```
> I chose Barack Obama, since he was president while I was growing up and in my teens. The words "American," "America," "jobs," "people," and "energy" pop up the most.

Co-Occurance
Co-occurrence gives us a sense of words that appear in the same text, but not necessarily next to each other. It shows words that are likely to co-occur. Note that this is different than topic modeling, which we'll discuss next week.

For this section we will make use of the `widyr` package. The function which helps us do this is the `pairwise_count()` function. It lets us count common pairs of words co-appearing within the same speech. This function might take a second as the resulting data frame will be incredibly large.

```{r}
sotu_word_pairs <- sotu_whole %>%
  mutate(speech_end = word(text, -5000, end = -1)) %>%  # extract last 100 words
  unnest_tokens(word, speech_end) %>%   # tokenize
  filter(!word %in% stop_words$word) %>%  # remove stopwords
  pairwise_count(word, doc_id, sort = TRUE, upper = FALSE) # don't include upper triangle
of matrix
head(sotu_word_pairs)
```

Now we have a list of words that appear near each other in the text as well as the frequency. Once again this dataset is far too large to look at in a data frame. Instead, we'll create a network graph that shows us the relationships between words for any words that appear more than 200 times. I chose 200 after looking at the above dataset and seeing that the highest count was 239. You want the network graph to be manageable and not too large.

```{r}
library(igraph)
library(ggraph)
sotu_word_pairs %>%
  filter(n >= 200) %>%  # only word pairs that occur 200 or more times
  graph_from_data_frame() %>% #convert to graph
  ggraph(layout = "fr") + # place nodes according to the force-directed algorithm of
Fruchterman and Reingold
  geom_edge_link(aes(edge_alpha = n, edge_width = n), edge_colour = "tomato") +
  geom_node_point(size = 5) +
  geom_node_text(aes(label = name), repel = TRUE,
                 point.padding = unit(0.2, "lines")) +
  theme_void()
```

(@) Create a network graph that shows the relationship between words that appear between 125 and 175 times.

```{r}
sotu_word_pairs %>%
  filter(n >= 125 & n <= 175) %>%
  graph_from_data_frame() %>%
  ggraph(layout = "fr") +
  geom_edge_link(aes(edge_width = n, edge_alpha = n), edge_colour = "blue") +
   geom_node_point(size = 5) +
  geom_node_text(aes(label = name), repel = TRUE, point.padding = unit(0.2, "lines"))

theme_void()

```

Analyzing Historical Journals

In the github repository below I have included the text and metadata for a journal called _Mind and Body_ which ran from the 1890s until the late 1930s and chronicled the development of the physical education profession. This dataset was OCR'd from copies stored in Google Books. Using the metadata provided and the raw text files can you use what you learned above to analyze these texts? What historical conclusions might you be able to draw?

```{r}
#zip file of all the .txt files. One for each issue.
```

````
unzip("MindAndBody.zip")
# Metadata that includes info about each issue.
```

(@) Add code chunks below and intersperse text to explain what you are doing and why.

```{r, setup, include=FALSE}

file_paths2 <- list.files("txt/")

mind.body <- readtext(paste("txt/", file_paths2, sep = "")) %>%
  mutate(doc_id = gsub("mb", "", doc_id))

mind.body <- full_join(mind.body, metadata, by = c("Filename" = "doc_id")) %>%
  as_tibble()

```



```{r}

tidy.dataset <- mind.body %>%
  unnest_tokens(word, text)

tidy.dataset

```

```{r}

tidy.dataset %>%
  count(word, sort = TRUE)

```

```{r}

stop_words.mb <- tibble(
  word = c(
    "st",
    "ft",
    "pa",
    "june",
    "july",
    "ave",
    "vol",
    "left"
  ),
  lexicon = "Mind Body"
)

all_stop_words <- stop_words %>%
  bind_rows(stop_words.mb)

tidy.mind.body <- tidy.dataset %>%
  anti_join(all_stop_words)
````

````
```

```{r}

tidy.mind.body %>%
  count(word, sort = TRUE)


```
````
````