Chapter 1: Counting Words in the 19th-Century British Parliamentary Debates

The introduction to text mining in this book begins with tracking words' context by processing data, counting words, and visualizing the words that co-occur with one-another. As the linguist John Rupert Firth (1974) said, "You shall know a word by the company it keeps." Tracking subtle variations in the context with which different words are used helps historians and other analysts to understand the cultural biases that structure our collective thinking. They may help us to understand how a culture understands men and women differently, how a culture describes people of different nationalities, the difference between a concept like "the foreign" and its near synonyms like "outsider" that nevertheless are used in important and different ways, or how these concepts are changing over time. In the course of this book, we will offer a taste of the subtle and refined approaches that analysts have used to unpack the context of different words.

Understanding the historical context of language begins with how we interpret and approach individual words. In this chapter, we treat the words we process as "keywords"–terms that are central to understanding culture and society, but whose meanings are often contested or have shifted over time (Williams 1985). For example, the word "labour" may be a particularly interesting word to an analyst of 19th-century Britain, more so than the word "table." This is because in the 19th-century debates, the meaning of the word "labour" evolved alongside industrialization, class conflict, and political reform. Early in the century, labour was often used in economic terms, referring to the input of human work in production, as discussed by political economists like Adam Smith. But as the century progressed—and especially with the rise of trade unions, Chartism, and later the Labour movement—the term labour took on a more political and moral dimension. It began to signify not just work, but the working class itself, with connotations of exploitation, rights, and collective struggle. The meaning of the word "table" on the other hand, is generally stable, literal, and uncontroversial. It refers to a physical object–typically a piece of furniture–and does not carry the kind of cultural significance that keywords do.

Recognizing words as evolving and changing–rather than fixed–allows us to glean greater insight into how language reflects broader cultural transformations. It invites us not to take words at face value, but to remember that language is dynamic and often shaped by power struggles, and that the way we process data impacts our subsequent analyses of these power struggles.

The data processing techniques we foreground will lead to our first visualization, where we transform individual utterances or debates into high-level visualizations depicting the frequencies of keywords and their associations (Moretti 2013, Jockers 2013). To achieve this end, we will introduce three basic techniques for processing text. We will: a) access data for the British parliamentary debates of the House of Lords and House of Commons, 1850-59 using the `hansardr` library, b) filter a relevant decade for keywords of interest, and c) count the words that appear in the same context as our keywords in order to study the associations of the keywords in more detail.

Importantly, the techniques presented in this chapter have both promise and limits. As our companion volume, *The Dangerous Art of Text Mining* (2023), makes clear, simple word counts are almost never sufficient on their own to support an in-depth analysis of culture.

Nevertheless, knowing how to count words and compare the context with which different concepts are used is a first step to using text as data. Recognizing that langauge contains keywords that can, for example, be chosen by an analyst to guide data processing is a first step towards producing meaning from text data.

Running a Line of Code

The first step in learning R is understanding how to run a line of code.

Here are some exercises to learn how to run code on your computer. Try each and decide which ones feel most comfortable to you: - You can type a command directly into the Console pane and press Enter. - You can write it in an R script and execute it by placing your curser on the line you want to run and press `Ctrl + Enter` on Windows or Linux (or `Cmd + Enter` on a Mac). - You can also highlight one or more lines of code and click the green Run button at the top of the script editor.

Try to run the code now by typing the following into your Console and pressing Enter, or by highlighting it and clicking Run:

```r
print("Hello World")
```

```
## [1] "Hello World"
```

This should display the phrase "Hello World" beneath your code, confirming that R executed the command.

Loading the Hansard Data

First we will load the data we wish to analyze. Analysts, programmers, researchers, and the like routinely import software packages developed by others to expand the range of tools available to them.

The authors developed a data package to go with this book, `hansardr`, to give readers an ability to access the Hansard 19th-Century British Parliamentary Debates within the R environment (Buongiorno 2022). This is a cleaned, analysis-ready corpus of the 19th-century British Parliamentary Debates (1803-1899), also known as "Hansard" in reference to Thomas Curzon Hansard, the publisher. It identifies debates whose records are missing from UK Parliament's corpus, and it also offers a field for disambiguated speakers. We believe this dataset will enable researchers to analyze the Hansard debates, including speaker discourse, in a way that has not been accessible before. Once installed, the data can be loaded with the `data()` function.

Functions are an important characteristic of the R language. Unlike other popular programming languages–such as Python–R is a functional language, which means it emphasizes processing data using these functions rather than relying on complex programming syntax. In R, a function is a block of code that performs a specific task. It can take inputs, called "arguments", that serve as instructions for how the function should behave, process data, or return an output. Functions are like the verbs of the R language in the sense that they say what should be done to the data. We will use functions throughout this book to perform data processing.

To use a function from another package, you must first install the package with `install.packages()`. Here is how you would install the packages used in this chapter with `install.packages()` before loading them with `library()`, which we demonstrate later in the chapter:

```
install.packages("tidyverse")
```

```
## Installing package into '/home/stephbuongiorno/R/x86_64-pc-linux-gnu-library/4.5'
## (as 'lib' is unspecified)
```

```
install.packages("tidytext")
```

```
## Installing package into '/home/stephbuongiorno/R/x86_64-pc-linux-gnu-library/4.5'
## (as 'lib' is unspecified)
```

```
install.packages("quanteda")
```

```
## Installing package into '/home/stephbuongiorno/R/x86_64-pc-linux-gnu-library/4.5'
## (as 'lib' is unspecified)
```

In RStudio, you can also verify whether a package is already installed by checking the Packages tab in the lower-right panel. This tab displays all packages currently available in your R library. If a package appears in that list, it is already installed; if not, you can install it using `install.packages()`. You can also load an installed package by checking the box next to its name in this tab, which is equivalent to calling `library()` in your script.

When installing code or data from a repository such as GitHub, we can use a special installation script that we created for this book.

If RStudio prompts you to restart R while installing a package, click Yes to allow the restart, and then re-run the installation command so the package loads into the fresh session.

```
source("https://raw.githubusercontent.com/Democracy-Lab/hansardr/master/tools/install_hansardr.R")
```

As we explained in the introduction, packages only have to be installed once, but they need to be reloaded for each new R session using `library()`. Only once we have run `library(hansardr)` will the data be available for our use.

```
# load the hansardr package
library(hansardr)
```

`hansardr` is now ready for use. It is conventional to load the software packages that will be use at the beginning of each new session. However, for the sake of demonstrating computational historical thinking, we will regularly load packages throughout a chapter so the reader can more easily see which functions relate back to different packages.

When working with functions in R, it is important to remember that code must be run in order, because each step often depends on what came before it. For example, if we plan to use a package such as hansardr, we must first load it with `library(hansardr)` before calling any of its functions. If we skip this step, R will not know where the hansardr functions come from and will return an error such as "could not find function," even if the code is correct. Similar errors can also occur when data is not processed in the proper sequence, since later steps may rely on datasets or data transformations that have not yet been created.

Navigating the `hansardr` Data

To make working with the Hansard corpus quicker and easier–and to prevent our personal computers from being overloaded with too much data at once–we partitioned the century long corpus into decade sections. In the field of data analysis, it is common to call these sections "subsets."

The first subset we will meet is a collection of the speeches of parliament, 1850-59, broken into sentences, where each row of the data frame is one sentence. Later, we will meet more datasets that contain "metadata," or information about the speaker, year and date of the speech, and other important contextual information for historical analysis.

The `hansardr` corpus is subsetted by decade. As shown by Table 1, each decade has four types of data, labeled: "hansard," "debate_metadata," "speaker_metadata," and "file_metadata." In the following table, "YYYY" stands in for any given decade.

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.1     v stringr   1.5.2
## v ggplot2   4.0.0     v tibble    3.3.0
## v lubridate 1.9.4     v tidyr     1.3.1
## v purrr     1.1.0
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter()     masks stats::filter()
## x dplyr::group_rows() masks kableExtra::group_rows()
## x dplyr::lag()        masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

Table 1: Table 1: Description of Hansard Corpus Files by Decade

Label	Description	Key
hansard_YYYY	Hansard debate text	sentence_id
debate_metadata_YYYY	Metadata such as speech date and title.	sentence_id
speaker_metadata_YYYY	Metadata on speakers.	sentence_id
file_metadata_YYYY	Metadata for source file, column, and more.	sentence_id

Both text and metadata are divided into subsets named by decade, for instance, `hansard_1850`. `hansard_1850` contains just the speech text as well as a unique sentence ID that can be used to connect the speech text with its corresponding metadata. We will demonstrate this process later in *Text Mining for Historical Analysis*.

Processing both text and dates is important for historical analysis. In this chapter, we will start working exclusively with text–not dates–because they are easy to process when counting words. But the other subsets within `hansardr` contain other data types.

We can use `data()` to load one of these decade-based subsets of the Hansard corpus.

```
# load the Hansard debate text for 1850
data("hansard_1850")
```

A new variable named `hansard_1850` is now able to be seen in our Global Environment panel. If you are using RStudio, you can explore the data by clicking on its name in the Global

Environment pane (the top right quadrant of RStudio). Clicking on the name of the dataset will open a new tab containing a view of the data, which can be searched for a keyword, like in a spreadsheet. One can also double-click on any row in the data frame to select a row, then copy that data and paste it into a Word Document or other tool.

The Hansard subsets are purposefully small enough to open and explore on a personal computer. When working with larger datasets, it may not be possible to open the entire dataset in another tab. Instead, in the Global Environment Panel you can also click on the blue ">" symbol to the left of each data frame to display the "fields" by which the dataset is structured, in this case, the names of the columns in the dataset, as well as a preview of the content of each of these columns.

In addition to exploring the data in RStudio, there are a few functions that provide the means to look at sections of the dataset in greater detail. Throughout this book, we will routinely "inspect" the data by using functions like `head()` to see how the data is transformed at different processing steps. Getting into the habit of using functions like `head()` makes it easier to notice and interrogate problems as they arise, or ensure that the data transformations meet expectations. At the console, an analyst can also use `head()` to ask R to display the first six rows of the data. In some versions of RStudio you will see an arrow that allows you to move between two columns.

As shown by Table 2, the first column in this dataset is `sentence_id`, which gives a unique identifier for each sentence, including information about which series of parliamentary proceedings we are looking at, which volume and page, and which sentence number on each page. The second column is `text`. Each row is a distinct sentence of text. These speeches were broken into sentences during the dataset creation stage, where the authors wrote a script to detect punctuation and line breaks.

Below is an excerpt from `hansardr`:

Tidy Text Mining and Loading Libraries

Most of the functions used in the first chapters of this book come from a software package called `tidyverse` (2019) which is a family of smaller packages developed by Chief Scientist at Posit (formerly known as RStudio), Hadley Wickham.

The tidyverse was created to order data in "tidy" tables where it is structured as a data frame made up of columns and rows. In other words, the tidyverse is built to work with data like `hansard_1850`.

We have found that a tidy approach to text exploration and analysis allows us to write code quickly and effectively, giving us ample control over the data and allowing us to inspect the output of every data processing step and type of analysis. This library thus supports analysts focusing on questions about data processing and history without being weighed down by the nuances of different structural and syntactic coding problems.

We distinguish between writing code and processing data, proposing that data processing and historical analysis are co-creative practices. Data processing shapes how we interpret the past—it is not a neutral task, but one that governs the possibilities of historical understanding. At the same time, the historical material we study–the observations we make–might inform and reshape our data processing methods. We will return to this central idea of "computational historical thinking"–which we introduce in our introduction–throughout the book, as we demonstrate how iterative adjustments to our processing approach impact our analysis.

In this chapter we will also use the `tidytext` library, developed by David Robinson and Julia Silge (2016), to work with textual data. The `tidytext` library contains a variety of tools useful

Table 2: Excerpt from Hansard 1850

sentence_id	text
S3V0108P0_0	, which had been prorogued successively from the 1st of August to the 9th of October, from thence to the 20th of November, thence to the 16th of January, and from thence to the 31st January, met this day for despatch of business.
S3V0108P0_1	The Parliament was opened by Commission, the LORDS COMMISSIONERS present being the LORD CHANCELLOR; the LORD PRESIDENT OF THE COUNCIL (the Marquess of Lansdowne); the LORD PRIVY SEAL (the Earl of Minto); the LORD CHAMBERLAIN OF THE HOUSEHOLD (the Marquess of Breadalbane); and the LORD BISHOP OF LONDON.
S3V0108P0_2	called attention to a great omission of their duty on the part of Ministers, with respect to the privileges of their Lordships, which might and ought to have been avoided.
S3V0108P0_3	At pre-sent there were two vacancies in the representative Peers of Scotland, in consequence of the deaths of the Earl of Airlie and of Lord Colville.
S3V0108P0_4	Although the Act of Parliament directed that the proclamation should issue forthwith for the election of representative Peers to fill up any vacancies which might occur, by death or otherwise, no such proclamation had yet taken place in the case of the two vacancies he had just mentioned, and the consequence was, that for twenty-four days after the meeting of Parliament it was not possible for any Scotch representative Peers to be elected.

for processing text and treating text as data. Many sections in this book will therefore begin by loading the two libraries `tidyverse` and `tidytext`.

```r
# load the tidyverse and tidytext packages
library(tidyverse)
library(tidytext)
```

Using Functions to Process Data

The primary purpose of a function is to create reusable code, allowing analysts to avoid repeating the same data processing steps over and over.

From this point on we are going to use more complicated functions from different R packages to process our data. Before we take this next step, however, we first want to provide a basic understanding of a function's underlying components, by creating our own simple function:

```r
# define a custom function to add two numbers
add_numbers <- function(a, b) {
  result <- a + b
  return(result) }
```

The above code can be broken down into these core parts:

1. Function Name: The name given to the function. In this example, the function is named `add_numbers`.
2. Parameters: The variables listed in the function definition that will accept arguments, for instance, `function(a, b)`.
3. Body: The code that is executed when the function is called, written here as `result <- a + b`.
4. Return Value: The output produced by the function, for example, `return(result)`.

Every function in R encapsulates a code body, even when it is hidden from the analyst.

We can now use our new `add_numbers()` function by passing arguments to the specified parameters. In this example, the "a" parameter is passed the number 2, and the "b" parameter is passed the number 3. The two numbers are added in the body, and then the results are returned.

Now that we have defined this function, we can call upon it whenever we need to add two numbers without rewriting the steps involved. This kind of reuse becomes especially important in historical analysis, which is often iterative and rarely linear. As our interpretations shift and our questions evolve, we often return to the same operations with new purposes. Having stable tools like functions allows us to revisit and revise our analyses without starting from scratch each time.

```r
# use the new function we created to add 2 and 3
print(add_numbers(2, 3))
```

```
## [1] 5
```

To count words in Hansard, we will use a different function, the `unnest_tokens()` function from `tidytext`. While there are many ways to count words over time, one way to do this–that is also very human readable–is to put our data into a format where each word of the text is on a separate row. These individual words are called “tokens.” The process of splitting chunks of text into individual units is called “tokenization.” While singular words are often called “tokens,” a common term used to instead describe multi-word phrases is “n-gram” (such as using the term “bigrams” to describe sequential, two-word phrases).

In this one-token-per-row format we can easily process our data. This approach to data processing is common in the humanities and handles text using the “Bags of Words” (BoW) approach, where text data is treated as a collection (or “bag”) of individual words without regard to grammar, order, or context (Jockers 2015). The focus is instead on word frequency or occurrence.

```r
# tokenize the 'text' column of the hansard_1850 dataset into individual words
# - 'unnest_tokens()' from the 'tidytext' package splits text into lowercase words
# - the new column will be called 'word'
# - each word becomes its own row
tokenized_hansard <- hansard_1850 %>% # create a new dataframe
  unnest_tokens(word, text) # tokenize the 'text' column
```

The above code also has new symbols, called “operators.” Operators are essential for data processing. Throughout the course of this book we will introduce multiple different kinds of operators.

The assignment operator, `<-`, sometimes translated as “gets,” points to the output of a function and is followed by the name of an original dataset. The `%>%` operator, called a “pipe,” and translated as “next,” is used to chain together successive transformations of a dataset. This will make more sense when we cover more examples later in the book.

If this is your first time using a programming language for analysis, it can be useful to practice reading each line of code as a sentence of instructions that tells the computer what output to make, what input to use, and what processes to go through in a certain order. We sometimes suggest that first time programmers translate code into a series of English sentences, using their knowledge of the `<-` and `%>%` operators, the names of variables, and their understanding of distinct commands. The code above be read as follows: “Create a new variable called `tokenized_hansard`. Start with the dataset `hansard_1850`. Next, unnest the tokens in the”text” column of `hansard_1850`, unpacking those strings of text into individual words.”

One may also notice the arguments passed to `unnest_tokens()`, “word” and “text.” Functions like `unnest_tokens()` take a set number of arguments, typically in a precise order. The first argument is the type of unnesting that will be employed. The second argument is the name of the column from the original `hansard_1850` data that we will work on, in this case, “text.”

As you inspect the arguments of each function, consider the fact that those arguments exist because they allow the analyst to change the arguments to get different results. With the `unnest_tokens()` function, the analyst could hypothetically choose to “unnest” the sentences into tokens of different length, for instance, sentences, n-grams, or other units of text. In later chapters we will explore some of these capabilities of the `unnest_tokens()` function. Alternatively, if `unnest_tokens()` were being applied to a different dataset where the column had a different name than “text,” we could change the column name from “text” to something else. In later examples, we will use the same commands applied to different datasets with different column names. For now, we will mainly use `unnest_tokens()` with the arguments “word” and “text.”

Next, let’s inspect the results of our unnesting using `head()`, which returns just the first rows of a data frame.

```r
# show the first few rows of our tokenized data
head(tokenized_hansard)
```

```
##   sentence_id         word
## 1 S3V0108P0_0        which
## 2 S3V0108P0_0          had
## 3 S3V0108P0_0         been
## 4 S3V0108P0_0    prorogued
## 5 S3V0108P0_0 successively
## 6 S3V0108P0_0         from
```

However, we have a preference for looking at more data since that can give us a fuller picture of the data.

```r
# show the first fifteen rows of our tokenized data
head(tokenized_hansard, n=15)
```

```
##    sentence_id         word
## 1  S3V0108P0_0        which
## 2  S3V0108P0_0          had
## 3  S3V0108P0_0         been
## 4  S3V0108P0_0    prorogued
## 5  S3V0108P0_0 successively
## 6  S3V0108P0_0         from
## 7  S3V0108P0_0          the
## 8  S3V0108P0_0          1st
## 9  S3V0108P0_0           of
## 10 S3V0108P0_0       august
## 11 S3V0108P0_0           to
## 12 S3V0108P0_0          the
## 13 S3V0108P0_0          9th
## 14 S3V0108P0_0           of
## 15 S3V0108P0_0      october
```

`unnest_tokens()` did a lot of heavy lifting for us!

As part of our data transformation, the output data has a column named "word" for each individual token of the corpus. Notice that the words in `tokenized_hansard` look just like the words in the first sentence of the first speech in `hansard_1850`. Those speeches have been split into individual words, but the words are still in the same order. Notice that the command `unnest_tokens()` has also transformed all the words to lower-case and removed punctuation marks such as commas and periods. This transformation, often called "data cleaning" will make the words easier to count because the computer will not treat words differently based on capitalization or the presence of symbols.

As we inspect the results of unnesting, one might note that the input to `unnest_tokens()` was "tidy" in format – that is, it had two columns, one for "sentence_id" and one for "text". After working with `unnest_tokens()`, the resulting output, `tokenized_hansard`, also is "tidy" in format. That is, it has two columns, one for `sentence_id` and one for `word`. This tidy format is both easily human readable and also easy to process and count.

Counting Words

Now we are going to bring together what we have learned by chaining together multiple functions that will be used to process the Hansard data.

The `count()` function from the `tidyverse` groups like tokens with like tokens, then counts them. Below, we are passing `count()` one argument, the name of the column to operate on–in this case, "word" from the tokenized hansard data.

```
# count the frequency of each word in the tokenized dataset
# this creates a data frame with one row per unique word and a column 'n' for the count
tokenized_hansard_counts <- tokenized_hansard %>% # Create a new data frame
  count(word) %>% # count how many times each word appears
  arrange(desc(n)) # sort the results in descending order of frequency
```

With the command `arrange(desc(n))`, we tell the computer to arrange the results from greatest to least. We perform this step so that the results are easier to read, not because this step is necessary for counting words.

```
##   word       n
## 1  the 2588864
## 2   of 1404831
## 3   to 1126226
## 4 that  726686
## 5  and  706117
## 6   in  597753
```

We now have a frequency count of every word recorded in the speeches of Hansard, 1850-59. The initial results are unimpressive. In our dataset, as with most textual datasets, the most common words are also often the least interesting. For this reason, analysts compile lists of the most common words, called "stop words." Analysts often remove stop words from their results before analysis.

The `tidytext` package comes with a prepared list of stop words. We can remove them from our results by loading the "stop_words" dataset and then passing the dataset as an argument to the `anti_join()` function. `anti_join()` keeps all the rows in the first dataset that do not have a match in the second dataset. Using `anti_join()` we can remove words we do not want in our analysis from our data.

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

```
head(stopworded_hansard_counts)
```

```
##         word      n
## 1        hon 125465
## 2      house 118861
## 3 government  95417
## 4       bill  77322
## 5      noble  73180
## 6       lord  68942
```

These results cleaned of their stop words seem sensible for parliament. Some of them demonstrate the way that members of parliamentarians refer to each other, for example, as "my noble lord" or "the hon[ourable] member [in question]." Speakers in parliament also routinely refer to the business of parliament, especially referencing the “house [of commons or lords],” the “bill” or “question” under debate, the state of the “country,” and how much “time” is available for debate.

General words, of course, give us little insight into the business of parliament. Later sections will give us richer tools for examining how the content of speeches changed over time. For now, we have successfully achieved the first step of analysis, the processing of text into words, and the counting of words.

Searching for Keywords

The `tidyverse` package provides several commands that we will use throughout this book to track individual keywords. The `filter()` function allows us to find an exact match within the data. Suppose that we want to inspect the tenth sentence of the parliamentary speeches, series 3, volume 108, which has the id number “S3V0108P0_10.” Alternately, `filter()` can be used to subset the `stopworded_hansard_counts` data frame, returning only the rows where the word column contains the value “india”.

```
# filter the word column for "india"
stopworded_hansard_counts %>%
  filter(word == "india")
```

```
##    word     n
## 1 india 20816
```

In this code, we pass `filter()` two arguments: first, the name of the column we will search for (“word”), and second the name to match (“india”). Here we are using the == operator to indicate we want to return an exact match, as opposed to returning words that might contain the word “india” such as “indian”.

We can do the same for any other keyword of interest.

```
# filter the word column for "france"
stopworded_hansard_counts %>%
  filter(word == "france")
```

```
##     word    n
## 1 france 7146
```

```
# filter the word column for "jamaica"
stopworded_hansard_counts %>%
  filter(word == "jamaica")
```

```
##      word   n
## 1 jamaica 475
```

Filtering for every word we wish to track, one-by-one, can be tedious and time consuming. Filter can also be used with other R operators, such as `%in%`, that save us time and energy by giving us the opportunity to cycle through multiple words of interest instead of just one.

To do this, we can first create our own custom list of words we wish to analyze.

```r
# create a list of words commonly used to identify or refer to women
# the purpose is to help filter or analyze texts for references to women
identifiers_for_woman = c("girl", "woman", "girls", "women", "wife", "wives", "widow",
                          "widows", "sister", "sisters", "female", "females", "grandmother",
                          "grandmothers", "aunt", "aunts")
```

Now we can use the `%in%` operator to check if a word from our list, `identifiers_for_woman`, is in the "word" column.

```r
# filter the word counts to keep only those words that identify or refer to women
# this checks if each word is in the 'identifiers_for_woman' list
stopworded_hansard_counts %>%
  filter(word %in% identifiers_for_woman)
```

```
##           word    n
## 1         wife 2222
## 2       sister 1153
## 3        woman 1063
## 4        women 1062
## 5        wives  479
## 6        widow  315
## 7       female  294
## 8       widows  254
## 9      sisters  240
## 10     females  195
## 11       girls  180
## 12        girl   76
## 13        aunt   62
## 14 grandmother   18
## 15       aunts   10
```

`filter()` is useful for more than just text data. It also works on quantitative data with comparison operators like the greater than, >, or less than, <, signs. This is handy if we want to filter for words that are said more or less than a specified number of times, as one example. For the sake of readability, we are only going to show the first 20 words that follow our filter.

```r
# filter the word counts to include only words that appear fewer than 474 times
stopworded_hansard_counts %>%
  filter(n < 474) %>% # filter for words stated less than 474 times
  slice_max(order_by = n, n = 20) # select the 20 most frequent words
```

```
##          word   n
## 1    contrast 473
## 2      pardon 473
## 3    thirteen 473
## 4   elizabeth 472
## 5    laboured 472
## 6       renew 472
## 7        seal 472
## 8     trustee 472
## 9          19 471
## 10    compare 471
## 11   concerns 471
## 12  denounced 471
## 13  mortality 471
## 14   publicly 471
## 15    silence 471
## 16    uttered 471
## 17      pride 470
## 18 sacrifices 470
## 19 acceptance 469
## 20    secrecy 469
```

We can also use multiple operators in conjunction with one-another. In the following code we do this by supplying the `&` operator to combine multiple conditions. The following example filters rows where n is greater than 474 and less than 476.

```
stopworded_hansard_counts %>%
  filter(n > 474 & n < 476) %>% # keep words that occur exactly 475 times
  slice_max(order_by = n, n = 20) # select 20 words (all will have n = 475)
```

```
##           word   n
## 1       cities 475
## 2   depositors 475
## 3      jamaica 475
## 4        pause 475
## 5       shared 475
## 6 unfavourable 475
```

In just this quick glimpse of our data, meaningful words are beginning to appear. The word "Jamaica," for example, appears along with the word "depositors." These words hint at the political, economic, and social issues of the time, specifically the declining state of the economy. Members of Parliament discussed the slave revolt in Jamaica, including its scale and significance. Since the 17th century, Jamaica had its own parliament but was placed under martial law during the revolt. Its appearance here demonstrates that Jamaica was significant enough to command parliamentary attention, elevating the revolt to a major historical event that could not be ignored by Parliament.

Tailoring our Word Count Analysis

Detecting and filtering for keywords of interest can be particularly meaningful for using data to tell history.

Our next function, `str_detect()`, treats data as patterns and detects their presence or absence. It returns `TRUE` if the pattern is detected, or `FALSE` if it is not. When a data type represents only two possible values, `TRUE` or `FALSE`, it is typically referred to as a "Boolean" value.

In the following example, we iterate through each word in the `identifiers_for_woman` list to check whether the value contains "girl."

```
# check which words in 'identifiers_for_woman' contain "girl"
identifiers_for_woman %>%
  str_detect("girl")
```

```
##  [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE
```

The results from `str_detect()` are not very telling for analysis. Instead, for a textual analysis we often use both functions, `filter()` and `str_detect()`, together so that we return a more human-readable result.

The following example searches `hansard_1850` for the keyword "coal." Notice that `str_detect()` is nested inside the command `filter()`. Together, these commands tell the computer to filter for only those rows of the column "text" that contain the keyword "coal."

```
# filter speeches that mention the exact word "coal"
# \\b ensures "coal" is matched as a whole word, not as part of another word (e.g. "coalesce")
hansard_coal <- hansard_1850 %>% # create a new data frame
  filter(str_detect(text, "\\bcoal\\b")) # filter the text
```

Something else interesting is happening in our code. The word "coal" has a funny looking set of characters, `\\b`, on either side. Together with the word "coal," these special characters are make up part of a "regular expression" often abbreviated as "regex." Regular expressions are sequences of characters that define search patterns, typically used for string matching, processing, and extraction.

One reason the `str_detect()` function is so special is because, unlike `filter()` by itself, the `str_detect()` function enables us to match with regular expressions, not just words. `\\b` indicates a word's boundaries. It matches the position between a word character (e.g., letters, numbers) and a non-word character (e.g., space, punctuation, or the start/end of a string). `\\b` at the end ensures that "coal" is a standalone word, meaning it will match "coal" but not "coalesce" or "charcoal."

The result of our search is variable, `hansard_coal`, that contains only sentences from the 1850 debates that contain the word "coal."

```
head(hansard_coal$text)
```

```
## [1] "The manufacturers of this country relied on their capital, and their command of coal and iron, 
## [2] "Mines, also, except coal, clay pits, slate quarries, and other descriptions even of real propert
## [3] "One class of offences which was more particularly adverted to in this part of the Bill, was tha
## [4] "A coal-owner told him that he was plundered of many tons of coal per week, owing to the difficul
## [5] "They were subjected to great loss by the theft of small articles of coal, and they submitted fr
## [6] "Here, perhaps, is one county abounding in coal and stone-the latter advantageous for the constru
```

Finding and Comparing Keywords' Context

Typically, we want to do more than simply search for a singular keyword. We may also want to search for collocates, or words that are "co-located" next to one-another. These words form the linguistic context of a given keyword.

We already have everything we need to make this happen: we can search for a keyword, we can break up text into words, and we can count the words. We can use our knowledge to search for the context of several keywords and compare them in visual form.

The following code chains multiple data processing steps together.

```r
# tokenize the 'text' column into individual lowercase words and store them in a new column 'word'
# remove common stop words (e.g., 'the', 'and') that carry little meaning
# filter out any tokens that contain digits (e.g., years, numbers)
# count the frequency of each remaining word
# arrange the word counts in descending order for easier interpretation
coal_context <- hansard_coal %>% # create a new data frame
  unnest_tokens(word, text) %>% # tokenize text
  anti_join(stop_words) %>% # remove stop words
  filter(!str_detect(word, "[:digit:]")) %>% # remove tokens with digits
  count(word) %>% # count tokens
  arrange(desc(n)) # sort by frequency
```

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

Instead of running these functions independently, we have chosen to chain them together with the pipe, `%>%`, operator. Using the pipe allows us to use many functions sequentially without storing intermediary variables.

Notice the line of code that reads `filter(!str_detect(word, "[:digit:]"))`. `[:digit:]` is a regular expression in R that matches all digits, this way we do not have to specify them one-by-one. We added `!` to filter out any entries that comprise digits. The negation operator, `!`, tells R to do the opposite of what the function usually does. Using it in the `filter()` function and preceding `str_detect()`, we ask to return text, not digits like the dates recorded in Hansard.

```r
head(coal_context)
```

```
##         word   n
## 1       coal 422
## 2      steam  50
## 3     duties  46
## 4       bill  42
## 5 government  40
## 6      mines  40
```

These are the words that appear most frequently with the word "coal." When sharing our results, however, we may want to visualize our data instead of providing a table so we can analyze collocates more easily.

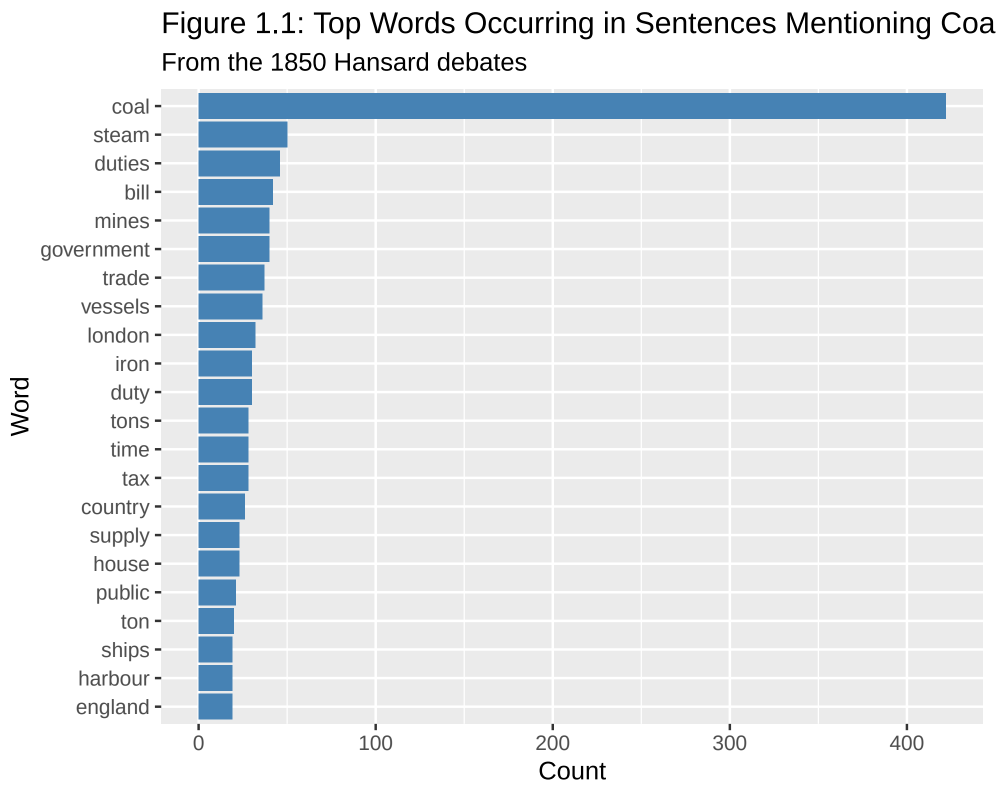
We can visualize just the top 20 words that co-occur with the word "coal" by selecting the `top_n(20)` appearances from our sorted data.

To visualize our data, we will use `ggplot`. `ggplot` is an R package used for creating visualizations. It follows an approach that allows users to build plots layer by layer, combining data, aesthetics (e.g., axes, colors), and geometric objects (e.g., points, lines) to create visualizations. We will use `ggplot` frequently throughout this book.

```r
# select the top 20 most frequent words from the 'coal_context' dataset
# this will be used to limit the plot to the most relevant terms
top_coal <- coal_context %>%
  top_n(20)  # top 20 words
```

```
## Selecting by n
```

```r
# create a horizontal bar chart of word frequencies
# this visualizes the most common terms in coal-related sentences
ggplot(data = top_coal) +
  geom_col(aes(x = reorder(word, n), y = n), # bar chart with reordered x-axis
           fill = "steel blue") + # set bar color
  coord_flip() + # flip for horizontal layout
  labs(title = "Figure 1.1: Top Words Occurring in Sentences Mentioning Coal", # add title
       subtitle = "From the 1850 Hansard debates", # add subtitle
       x = "Word", # x-axis label
       y = "Count") # y-axis label
```



The resulting bar chart shows us the number of times “coal” was mentioned, as well as the top words that appear with coal. Note that because we are analyzing collocates, these counts do not reflect the total number of times they were mentioned between 1850 to 1859. Instead, the counts represent the total number of times these words appear in the same context as “coal.”

The second most frequent word is “steam” followed by “duties,” “bill,” and “mines.” Words like “duty,” “iron,” “tax,” and “supply” indicate key economic and industrial topics related to coal during the debates, with “iron” possibly tied to the role of coal in iron production. “Harbour,” “ships,” and “vessels” suggest discussions related to transportation and trade of coal.

Rather than focusing on specific speeches, this method of visualizing data allows us to trace broader semantic and thematic structures that shaped the discourse around the keyword coal in the 1850s. By aggregating linguistic data over time, we can detect not only the persistence of industrial and economic concerns, but also a macroscopic view of historical language.

Taken together, the visualization emphasizes the central role of coal in the British economy and legislation in the 1850s, as well as its links to other industrial and governmental matters.

The visualization also reflects concerns around infrastructure, taxation, and policy decisions regarding coal. Our analysis will gain nuance the more we compare near concepts. While coal was important to powering Britain's industrial revolution, perhaps the most controversial economic debates of the 1850s revolved around the issue of the Corn Laws, the system of taxation of wheat and other grains, which were originally introduced to protect British farmers. Working-class demands for cheap bread led to the repeal of the Corn Laws in 1846. Knowledge of these facts might lead us to ask: were coal and corn debated using the same language or different words?

```r
# filter the dataset to include only rows where the word "corn" appears as a whole word
# tokenize the filtered text into individual lowercase words
# remove stop words that carry little semantic weight
# exclude any tokens containing digits (e.g., "1850", "2nd")
# count the frequency of each remaining word
# sort the words in descending order of frequency
corn_context <- hansard_1850 %>% # create a new data frame
  filter(str_detect(text, "\\bcorn\\b")) %>% # filter rows with 'corn' as a whole word
  unnest_tokens(word, text) %>% # tokenize text
  anti_join(stop_words) %>% # remove stop words
  filter(!str_detect(word, "[:digit:]")) %>% # remove tokens with digits
  count(word) %>% # count word frequency
  arrange(desc(n)) # sort by frequency
```

```r
head(corn_context)
```

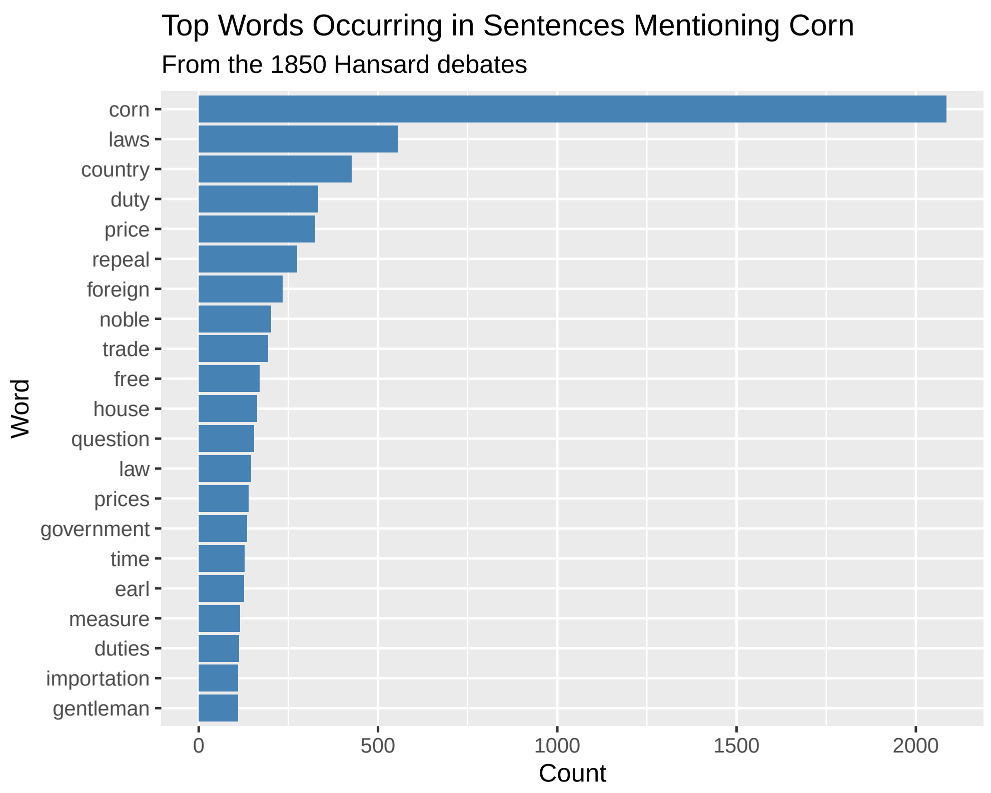
```
##      word    n
## 1    corn 2085
## 2    laws  556
## 3 country  426
## 4    duty  333
## 5   price  325
## 6  repeal  274
```

Already we can see the words that most frequently co-occurred with our keyword "corn." Visualizing the top 20 words using `ggplot` makes this context easier to apprehend.

```r
# select the 20 most frequent words from the corn-related word counts
# this subset will be used to visualize the most relevant terms
top_corn <- corn_context %>%
  top_n(20)  # top 20 words
```

```
## Selecting by n
```

```r
# create a horizontal bar plot showing word frequencies
# this plot highlights the most common words in sentences mentioning "corn"
ggplot(data = top_corn) +
  geom_col(aes(x = reorder(word, n), y = n), # draw bars sorted by count
           fill = "steel blue") + # set bar color
  coord_flip() + # flip axes for horizontal bars
  labs(title = "Top Words Occurring in Sentences Mentioning Corn", # add title
       subtitle = "From the 1850 Hansard debates", # add subtitle
       x = "Word", # x-axis label
       y = "Count") # y-axis label
```



Comparison between the words used in the context of coal and corn gives us a foothold for beginning to think about how taxation was discussed in different contexts. For instance, the context of both keywords reference the role of "time", "duties", "taxes", and the "government", but debates over corn are more linked to the question of Britain's dependence "foreign" powers, whereas debates over coal are more crucially related to the condition of "England" itself.

Further research would be necessary for an analyst to decide whether comparing these two commodities as keywords is worthwhile, and which collocates offer insight. Nevertheless, at this point we have demonstrated a preliminary process with which most of the analyses in this book will start: by loading data, inspecting data, breaking that data into words, and counting those words. Later chapters will complicate this process, giving us more sophisticated approaches to understanding historical change and change over time.

This foundational workflow, loading, inspecting, tokenizing, and counting words, lays the groundwork for more nuanced textual inquiries. One such method, which begins to address the limitations of mere frequency counts, is the "Keywords in Context" (KWIC) approach.

Finding a Word's Context using "Keywords in Context" (KWIC)

In the 1950s, Jesuit scholar Father Roberto Busa partnered with IBM to create a machine-readable concordance of the works of Thomas Aquinas, known as the Corpus Thomisticum (Jones 2016). As a Jesuit scholar deeply immersed in Thomistic philosophy, Busa wanted to analyze the use and meaning of the Latin word "in"—a preposition with significant theological weight—in the complete works of Thomas Aquinas. These texts contained over 11 million words, making it virtually impossible to conduct such a study without first restructuring the material for analysis. However, the challenge he faced was not only the scale of the corpus, but the need to preserve the linguistic and philosophical nuance of Aquinas's writings.

Widely regarded as the inception of the digital humanities, Busa's work demonstrated the transformational potential of computing for humanistic inquiry. Busa's project marked a pivotal shift in textual scholarship by introducing computational methods to the humanities. By producing a machine-readable concordance of Thomas Aquinas's works, the project laid the foundation for techniques now central to digital textual analysis, including the use of KWIC (Keywords in Context) displays.

Concordance-style views have remained central to the study of language and meaning in the humanities. While collocates are one way to explore a word's context—by analyzing frequently co-occurring tokens—they are not the only method. Tools like "Keywords in Context" (KWIC), available in the `quanteda` library, provide a concordance-style view that presents sentence-level context. This allows researchers to examine not just which words appear near each other, but how a term functions within its surrounding textual context

To look at just the month of February, we need to load our decade subset that contains information about dates, called `debate_metadata_1850`, and join it with the dataset containing the debates, `hansard_1850`.

```r
# Load the debate metadata for 1850
data("debate_metadata_1850")
```

```r
head(debate_metadata_1850)
```

```
##   sentence_id speechdate                       debate
## 1 S3V0108P0_0 1850-01-31       MEETING OF PARLIAMENT.
## 2 S3V0108P0_1 1850-01-31       MEETING OF PARLIAMENT.
## 3 S3V0108P0_2 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 4 S3V0108P0_3 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 5 S3V0108P0_4 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 6 S3V0108P0_5 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
```

We will cover joins in greater detail in Chapter 2. For now it's just important to know that we are using our unique ID, `sentence_id`, to join the two data frames together so that the speeches are aligned with their correct dates.

```r
# Merge the hansard_1850 text data with its corresponding metadata
# using a left join to keep all rows from hansard_1850
hansard_1850_with_metadata <- left_join(hansard_1850, debate_metadata_1850)
```

```r
head(hansard_1850_with_metadata)
```

```
##   sentence_id
## 1 S3V0108P0_0
## 2 S3V0108P0_1
## 3 S3V0108P0_2
## 4 S3V0108P0_3
## 5 S3V0108P0_4
## 6 S3V0108P0_5
##
## 1
## 2
## 3
## 4
## 5 Although the Act of Parliament directed that the proclamation should issue forthwith for the electi
## 6
##   speechdate                       debate
## 1 1850-01-31       MEETING OF PARLIAMENT.
## 2 1850-01-31       MEETING OF PARLIAMENT.
## 3 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 4 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 5 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 6 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
```

In the following code we filter for speech dates that are on or after (greater than or equal to >=) February 1, 1850, and on or before (less than or equal to <=) February 28, 1850.

```r
# Filter the dataset to include only speeches from February 1850
top_month <- hansard_1850_with_metadata %>% # create a new data frame
  filter(speechdate >= "1850-02-01", # filter
         speechdate <= "1850-02-28")
```

KWIC returns a concordance-style text output with the words that come directly before and after the keyword in a sentence. Quanteda's version of `KWIC()` uses a different approach from `tidyverse`. It processes data quicker if we first transform the data from a tidy table into a "quanteda corpus" using `corpus()`. Using this approach, we can tell Quanteda that we wish to process the `text` column.

When using `KWIC()` we can specify the keyword, the maximum number of words to display before and after the keyword, and we can also tell `KWIC()` to do a case insensitive match so that we match with uppercase or lowercase instances of our keyword.

```r
# load the quanteda package
library(quanteda)
```

```
## Package version: 4.3.1
## Unicode version: 14.0
## ICU version: 70.1

## Parallel computing: disabled

## See https://quanteda.io for tutorials and examples.
```

```r
# convert the filtered february 1850 dataset into a quanteda corpus object
# specify that the 'text' column contains the actual speech content
my_corpus <- corpus(top_month, text_field = "text")

# search for instances of the word "corn" in the corpus
# extract 5 words before and after each occurrence
# make the search case-insensitive so it matches "corn", "Corn", etc.
corn_kwic <- kwic(tokens(my_corpus), # tokenize corpus
                  "corn", # keyword to search
                  window = 5, # number of words before/after
                  case_insensitive = TRUE) # match regardless of case
```

```r
head(corn_kwic)
```

```
## Keyword-in-context with 6 matches.
##   [text2032, 7]             cultivation of land, as | corn |
##  [text2036, 11]           south of Ireland dealt in | corn |
##  [text2038, 13] , deterred persons from cultivating | corn |
##   [text2536, 2]                                  on | corn |
##  [text2568, 23]         laws governing the price of | corn |
##  [text2578, 13]                  to the repeal of the | corn |
##
##  ground, being given up
##  , and were the pro
##  .
##  , would never have been
##  .
##  laws, for those laws
```

Critical Thinking With Collocates

When is collocate analysis the right method? Researchers of conceptual history frequently use collocates to understand the subtle differences between synonyms. For instance, historian Ruben Ros (2021) has investigated the way that Dutch newspapers in the nineteenth century began to use terms such as "foreign," "overseas," and "strange" in the process of constructing a narrative that increasingly linked the dangers of foreign influence and suspicion of ethnic minorities.

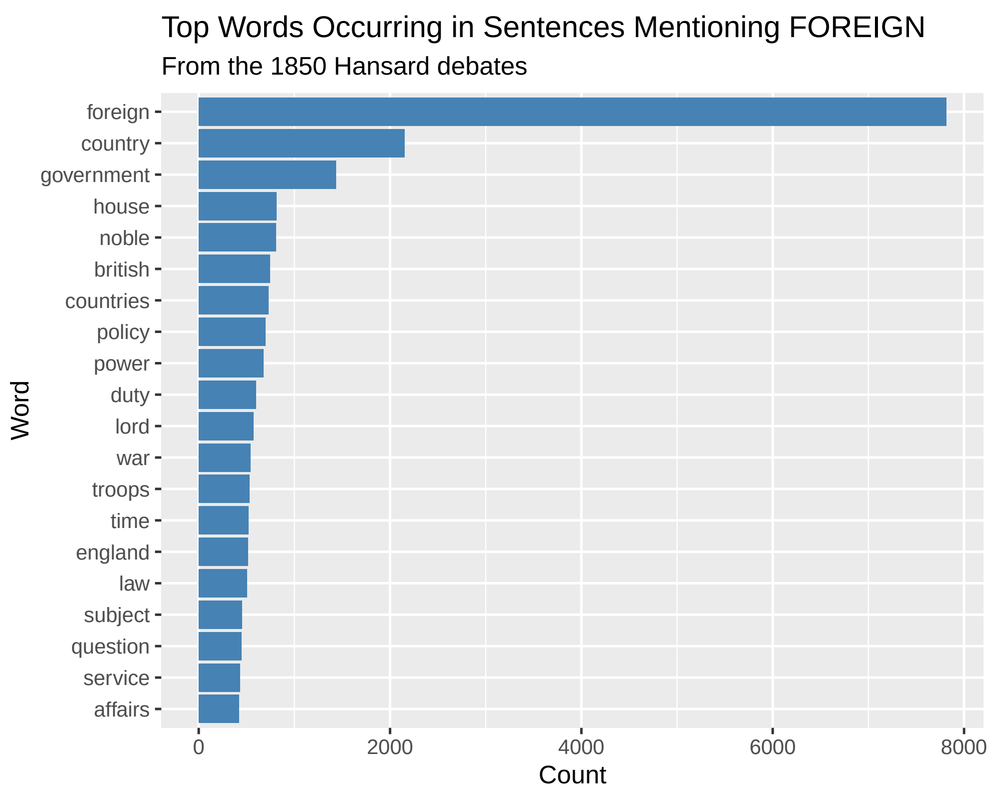
With collocate analysis alone, we cannot trace discourses over time – a crucial step in understanding the construction of political and cultural positions. But we can begin to tease apart the meanings of closely-related ideas at various points in the past. We can recreate Ro's approach in miniature here, asking the question: how did British members of parliament in the 1850s talk about foreigners and colonial subjects?

In the following code we create a function for filtering the Hansard data for a keyword, and visualizing the top 20 words. These are the same series of steps we have performed already, but here we are using our function to produce a series of related graphs in succession without writing out the same instructions multiple times.

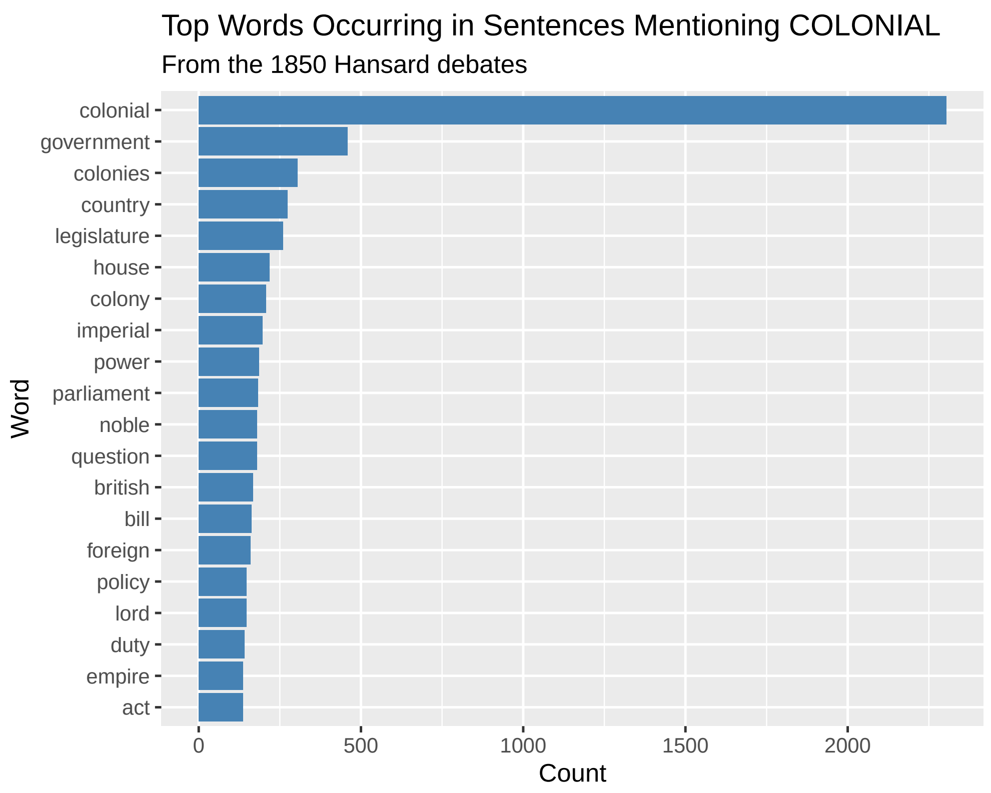
Creating a function will allow us to automatically create a series of graphs for the keywords we wish to analyze. Producing many graphs in succession is good practice for exploratory data analysis (EDA), the stage of research where we look at the data to understand its basic patterns, structure, and possible meanings before doing formal interpretation or modeling. Performing a comparative analysis of graphs side-by-side is one of the main skills digital humanists use when considering the subtle structures of meaning and difference that define cultural and political conversations. A careful comparison between the charts above offers the beginnings of a nuanced analysis of overlapping terms used by British members of parliament to refer to people and forces from beyond Britain.

```r
# define a function to generate a bar plot of top words that co-occur with a given keyword
# filters the dataset for rows containing the keyword as a whole word
# tokenizes the text, removes stop words and numeric tokens
# keeps the top 20 most frequent words and plots them as a horizontal bar chart
generate_word_plot <- function(keyword, hansard_data) {
    filtered_hansard_1850 <- hansard_data %>%
      filter(str_detect(text, paste0("\\b", keyword, "\\b"))) %>% # filter rows with keyword
      unnest_tokens(word, text) %>% # tokenize text
      anti_join(stop_words) %>% # remove stop words
      filter(!str_detect(word, "[:digit:]")) %>% # remove numeric tokens
      count(word) %>% # count word frequency
      top_n(20) # keep top 20 words

    plot <- ggplot(filtered_hansard_1850) + # create a new dataset for the plot
      geom_col(aes(x = reorder(word, n), y = n), # bar chart ordered by count
               fill = "steel blue") + # set bar color
      coord_flip() + # horizontal bars
      labs(title = paste("Top Words Occurring in Sentences Mentioning",
                         toupper(keyword)), # plot title
           subtitle = "From the 1850 Hansard debates", # plot subtitle
           x = "Word", # x-axis label
           y = "Count") # y-axis label

    print(plot) } # display the plot
```

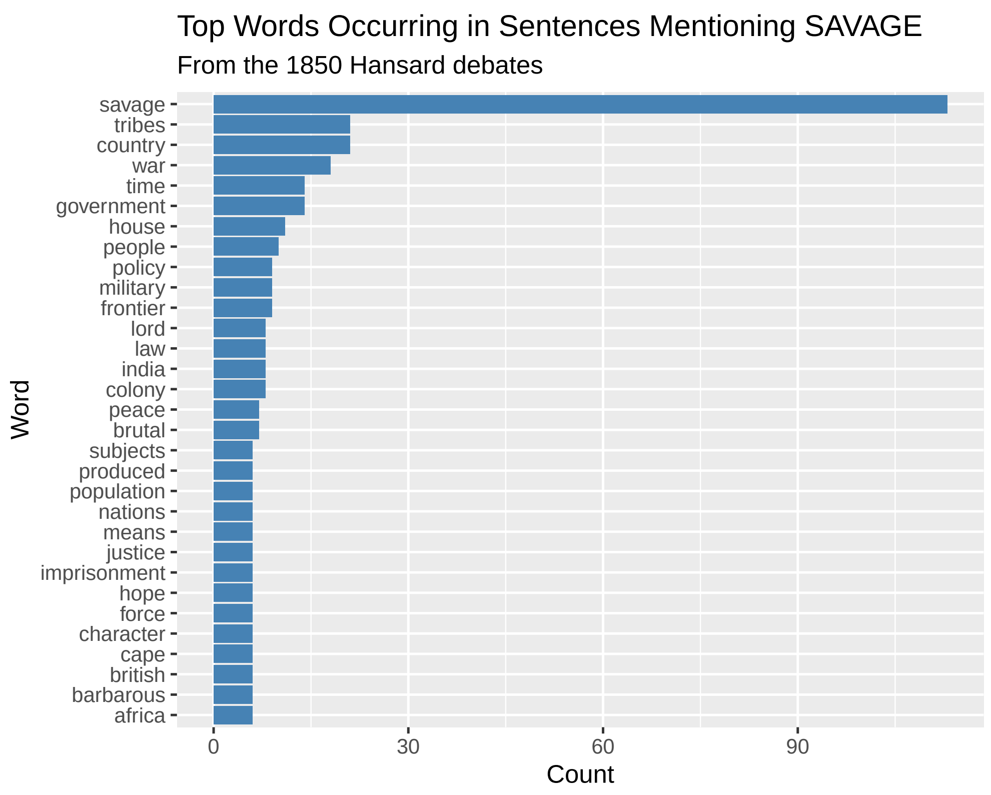
```
# use our function to create a clean bar plot of the top 20 words in sentences
# mentioning the word "foreign"
generate_word_plot("foreign", hansard_1850)
```



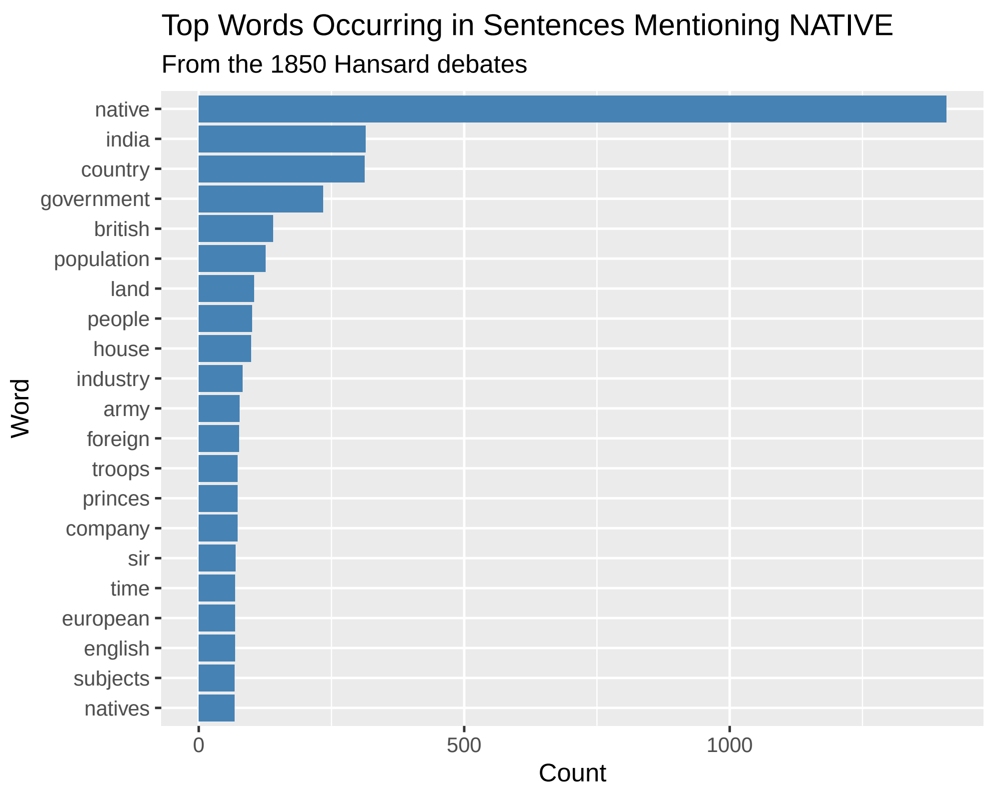
```
# do the same for sentences that mention the word "colonial"
generate_word_plot("colonial", hansard_1850)
```



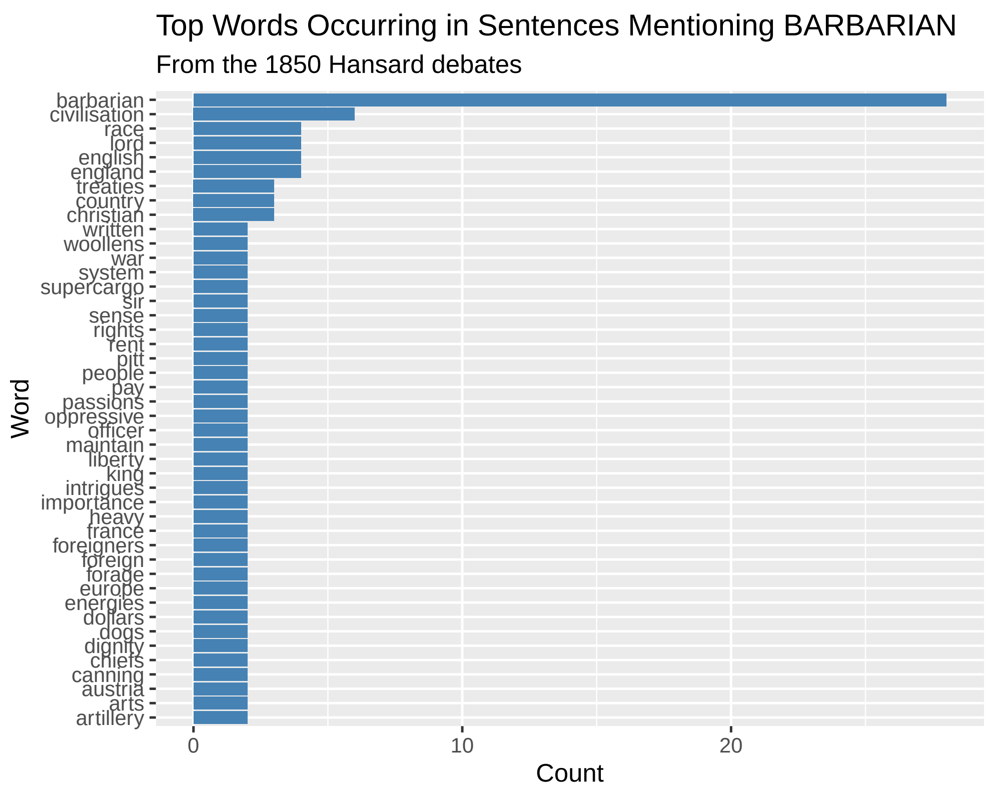
```
# do the same for sentences that mention the word "savage"
generate_word_plot("savage", hansard_1850)
```



```r
# do the same for sentences that mention the word "native"
generate_word_plot("native", hansard_1850)
```



```r
# do the same for sentences that mention the word "barbarian"
generate_word_plot("barbarian", hansard_1850)
```



Producing many graphs in succession can be useful for exploratory data analysis. Performing a comparative analysis of graphs side-by-side is one of the main skills digital humanists use when considering the subtle structures of meaning and difference that define cultural and political conversations. A careful comparison between the charts above offers the beginnings of a nuanced analysis of overlapping terms used by British members of parliament to refer to people and forces from beyond Britain.

In the results, we see several closely related terms which compose a field of interrelated meanings. Together, these terms sketch out the diversity of contrasting ways that British members of parliament spoke about the world outside of Britain. Some of the attributes ascribed to the foreign were neutral in value – for instance, the "foreign" was mainly spoken about in terms of "laws", "duties", "trade", and "protection", a world in which different nations "competed," but generally not a space of moral inferiority and superiority. By contrast, terms such as "savage" and "barbarian" telegraphed intellectual and ethical judgments (for instance, "brutal") onto cultures outside of Britain, typically associating inferiority with racial identity (Metcalf 1995, Colley 1992, Said 1987).

With these terms, a set of binaries is set up by the speakers of Parliament, dividing the world into "barbarian" cultures and "christian" "civilisations," a distinction that may have been analyzed in terms of "rights," "treaties," "war," "rents," "passions," "liberty," and "energies. The distinctions between the"foreign" and the "barbarian" are invoked in relationship to the same abstractions we saw typifying the "foreign," including "law," the "country," and "treaties." We also see the "barbarian" being invoked in discussions of many commodities, including "woolens," "forage," and "rent." Indeed, the frequent invocation of law, rights, and foreignness alongside discussions of the barbarian and the savage suggests that the judgments ascribed to race actually bled into conversations about law, tariffs, trade, and protectionism, perhaps as justification for invoking British superiority.

Some readers may assume that we are performing contemporary prejudices about racism, but the point of this exercise is that it is grounded in a detailed and careful examination of the words counted by computational methods. The paragraph above is not concocted out of thin air; it is an objective description of the prejudices on display in a quantitative reading of how British members of parliament spoke about the world beyond their nation. We emphasize that this analysis of the related language of the "foreign" and the "barbarian" is not concocted on the basis of our readings of contemporary theories about racism, but is actually derived from a careful reading of the language of Hansard, as well as immersion in writing about the history of empire.

In general, descriptions of arguments made on the basis of word counts tend to be more persuasive when they take on every word in a list. Critical thinking about the meaning of proximate keywords is enhanced when the analyst slows down, taking the trouble to examine the distinctiveness of each individual keyword and its context. In a further analysis, the scholar might ask questions that require examining the results in more detail. They might ask: What biases does each term convey? Where do the terms overlap? What new information is encapsulated by some words but not others? We could continue this line of inquiry to trace the keyword-context pairs back to their original context in sentences and speeches on the page, showing how speakers used these words to formulate actual arguments with consequences for the development of nations and their economies.

Examining collocates provides the basis for critical thinking about the changing and overlapping meaning of shared concepts in the human past. Skillful analysts can use approaches of this kind to examine the meaning of language about categories beyond those of gender, race, nationality, and commodities, asking questions about the intellectual categories that governed how members of parliament understood governance itself, economics, truth, science, virtue, economics, and reason itself. In every case, the approach would be the same: to examine not merely one or two keywords, but many related keywords, drawing attention to the subtle overlap and differences between the terms in usage, and how those overlapping concepts together work to produce a field of meaning.

Critical Thinking About Data and its Limits

Another fruitful avenue for critical thinking is the question of which problems are suited to textual analysis and which are not. Skillful analysts are wary of asking the wrong question with the wrong tool. Among the techniques that we have learned in this chapter is the ability to load and inspect data. Inspecting the data allows the analyst to notice issues caused by computational processing of text that may cause issues down the line unless the analyst is aware of them.

Looking at the first lines of `hansard_1850`, you might ask yourself these questions: are all of the lines in the "text" field actually sentences? Are all the words in the same style with respect to capitalization? We can inspect this further.

Analyzing the entire data frame to further inspect the data can be distracting since a data frame usually contains multiple columns, some which may be irrelevant to our current query.

```r
head(hansard_1850_with_metadata)
```

```
##   sentence_id
## 1 S3V0108P0_0
## 2 S3V0108P0_1
## 3 S3V0108P0_2
## 4 S3V0108P0_3
## 5 S3V0108P0_4
## 6 S3V0108P0_5
```

```
##
## 1
## 2
## 3
## 4
## 5 Although the Act of Parliament directed that the proclamation should issue forthwith for the elect
## 6
##    speechdate                        debate
## 1 1850-01-31       MEETING OF PARLIAMENT.
## 2 1850-01-31       MEETING OF PARLIAMENT.
## 3 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 4 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 5 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
## 6 1850-01-31 SCOTCH REPRESENTATIVE PEERS.
```

We can make the data more readable by displaying just the data from the column of interest.

In the following code, we use `hansard_1850[1:10, 2]` to select a specific item from `hansard_1850`. `hansard_1850` refers to a data frame or matrix, and `[1, 2]` refers to the row and column indices we wish to look at. The structure of the syntax is this: `[row, column]`. The numeric range, `1:10`, in the "row" field tells R to display rows 1 to 10. We put "2" in the "column" field to tell R to display just the second column, which in this case is the column containing text.

```
# view rows 1 to 10 from the 2nd column of the hansard_1850 dataset
hansard_1850[1:10, 2]
```

x
, which had been prorogued successively from the 1st of August to the 9th of October, from thence to the 20th of November, thence to the 16th of January, and from thence to the 31st January, met this day for despatch of business.
The Parliament was opened by Commission, the LORDS COMMISSIONERS present being the LORD CHANCELLOR; the LORD PRESIDENT OF THE COUNCIL (the Marquess of Lansdowne); the LORD PRIVY SEAL (the Earl of Minto); the LORD CHAMBERLAIN OF THE HOUSEHOLD (the Marquess of Breadalbane); and the LORD BISHOP OF LONDON.
called attention to a great omission of their duty on the part of Ministers, with respect to the privileges of their Lordships, which might and ought to have been avoided.
At pre-sent there were two vacancies in the representative Peers of Scotland, in consequence of the deaths of the Earl of Airlie and of Lord Colville.
Although the Act of Parliament directed that the proclamation should issue forthwith for the election of representative Peers to fill up any vacancies which might occur, by death or otherwise, no such proclamation had yet taken place in the case of the two vacancies he had just mentioned, and the consequence was, that for twenty-four days after the meeting of Parliament it was not possible for any Scotch representative Peers to be elected.
The Peerage of Scotland was not therefore represented in Parliament at present as it ought to be.
He wanted to know what his noble Friend the President of the Council had to urge in defence of this omission; for certain it was that the Act of Parliament had not been obeyed?
said, that the Government were not to blame for the omission, as they had not received any requisition from the usual quarter; they had strictly followed the previous usage.
said, the explanation of the noble Marquess was unsatisfactory.
The clause in 5 & 6 Anne, c. 8, directed what should be done in the case of vacancies by death, and the words could not well be misunderstood.

Some of the rows in the "text" field appear not to be complete sentences. We can investigate this oddity further.

```r
# view the first row from the 2nd column of the hansard_1850 dataset
hansard_1850[1, 2]
```

x
, which had been prorogued successively from the 1st of August to the 9th of October, from thence to the 20th of November, thence to the 16th of January, and from thence to the 31st January, met this day for despatch of business.

At this point the dataset is too broad to clearly display by printing, and some data is cut off. To view the whole text, double-click on the cell, copy and paste the text into a word processor. The content reads like so:

> ", which had been prorogued successively from the 1st of August to the 9th of October, from thence to the 20th of November, thence to the 16th of January, and from thence to the 31st January, met this day for despatch [sic] of business."

The first row appears to be a description of the opening of parliament. Technically, it is a commentary by the printer, not a line of debate. The convention of discussing parliament's timetables was in place in 1850, but we do not know if the printer always printed this line through the entire century. We should be aware of this convention and curious about it if we find ourselves counting months or discussions of words such as "business."

Let's keep inspecting the data, this time turning to the third row in our dataset.

```r
# view the third row from the 2nd column of the hansard_1850 dataset
hansard_1850[3, 2]
```

x
called attention to a great omission of their duty on the part of Ministers, with respect to the privileges of their Lordships, which might and ought to have been avoided.

Here, we see another sentence fragment:

> "called attention to a great omission of their duty on the part of Ministers, with respect to the privileges of their Lordships, which might and ought to have been avoided."

The sentence fragment has resulted from attempting to process data while working on another convention of printing: the fact that the publisher of Hansard for much of the century gave the name of the speaker followed by a description of the airing of the speech. On the page, we would be given the name of a speaker or their title, for example, "The Lord Chancellor" or "Mr. Gladstone." The passage of text would therefore read, [the speaker] "called attention to a great omission…" – in other words, it would not be a sentence fragment, so much as a journalistic description of a speaker. The actual speech probably began with a statement something like this: "It is a great omission of their duty on the part of Ministers[…]." In other words, we are looking at an oddity that resulted from a mismatch between the 19th-century printed record and how the computer processed the data. We compiled the data and separate all of the speaker names into one "field" of data, which we stored in a separate dataset, to be accessed in Chapter 2. We have sorted all of the text into the "text" field, which is held in `hansard_1850`. The computer did not process the journalistic description of a speech rather than a normal speech.

Looking at the data presents an opportunity for critical thinking. Analysts of text as data routinely need to inspect their data and make sure that the data's quality and density are sufficient to support claims that they might make.

Already, we can see that the data has certain features that might interfere with certain kinds of queries, and we should already be thinking about what these limits might be. Yet the convention of journalistic reporting on speeches can cause trouble for the analyst if the analysts are not aware of what has changed. The conventions of describing speeches changed over the course of the century; at the beginning of the century, journalistic descriptions were more common. Later in the century, the conventions changed, and many more speeches were reported near verbatim without journalistic commentary. We should be aware of conventions of this kind if we count words which may be related to journalistic observations such as "attention," "spoke," "declared," and so on. Otherwise we may be tempted to interpret changes to the publisher's conventions of printing speeches as evidence for changing ideas about politics.

Whether or not data quality will interfere with our ability to make inferences from the data depends on what we are asking. If we are counting mentions of the word "duty" or the rest of the words in the content of the speech, then our counts of the words will be correct. If we contrive to study the history of journalistic observation by tracking sentence fragments in the dataset, this data will support such a reading. But we should be wary making inferences about any kind of inquiry where there is a risk of overlap between substantive discussions and journalistic observation.

Exercises

Expand your grasp of the code and your powers of critical reflection by trying the following exercises:

1) Alter the code above to compare the context for the keywords "girl" and "woman." Do you expect the context to be similar for both? Does anything surprise you about the results?

2) Use your powers of inspecting data to move from the keyword context back to the individual sentences. Cut and paste twenty sentences from the context of each keyword into a word processor. What do you learn from these sentences that you did not learn from counting keywords? Does your analysis become more persuasive when quantitative graphs are accompanied with readings of particular speeches where the words are examined in their original context?

3) We have examined 'coal' and 'corn' in the 1850s. Now, search for the words 'coal' and 'corn' in dataset `hansard_1820`. Were they being used in the same way? What changed?

4) Above, we created the list of words called `identifiers_for_woman`. Use the operator `%in%` with the command `filter()` to create a list of the words in the context of discussions of women. Create another list called `identifiers_for_man` and do the same for discussions of men. Create visualizations for the top terms used in discussions of men and women.

5) Reflect on your work so far – in which you have compared words for gender, the concept of the foreign, and the context of discussions of different commodities. You have made