12

Text Data

So far in this book, we worked almost exclusively with data structured in tabular form. In the previous chapter, we saw how simple tables can be amended with spatial coordinates, such that each entry in the table is linked to a location on the globe. In this chapter, we cover a type of data with considerably less structure: texts. A text can be any written statement or report, but also spoken words that were transcribed. Text data is important to understand a wide variety of phenomena that are of interest to social scientists: What issues are discussed in parliamentary debates? How do certain hash tags travel on social media? How do journalists frame particular social issues in news reporting? All these applications require us to deal with text data.

In recent years, text analysis has become extremely popular in the social sciences. While traditionally the domain of (computational) linguistics, several types of text analysis have now become part of the social science toolkit. Linguists focus more on in-depth analysis of text where they try to identify, for example, the structure of sentences, and the subject or the object of an action. In the social sciences, and in political science in particular, text analysis has been done with simpler approaches, for example, analyzing word frequencies, or searching for the occurrence of particular keywords in texts.

Whatever type of analysis we plan to apply to textual data, we will have to obtain, manage and store these texts first. This is what we focus on in this chapter. The variety of methods for text analysis is so large that this chapter cannot even provide a sufficient overview. Rather, we discuss what text data look like, how they are typically stored as files, and how we can process and manage these data both in R and in a relational database.

Country	USA
Year	1978
Debate	34

We meet in this General Assembly on the threshold of a new decade. It will be a time of complex challenge, a period in which, more than ever, co-operative endeavours among nations are a matter not only of idealism but also of direct self-interest. [..]

Country	HUN
Year	1997
Debate	52

I am very pleased to see the Foreign Minister of neighbouring Ukraine assume the prestigious post of President of the General Assembly at its fifty-second session. In fulfilling your challenging tasks you may rest assured of the support and cooperation of the delegation of Hungary. [..]

FIGURE 12.1. Two documents with associated metadata.

In doing so, and in line with the general approach of this book, the focus is on the handling of text data *before* it is used for some kind of text analysis; therefore, we deal with the representation and storage of textual data, and how we can search and query them. Once we know how to do this, the data can later be used for all the different methods and tools that exist for text analysis, such as topic modeling, sentiment analysis or more advanced natural language processing approaches (Grimmer et al., 2022).

12.1 WHAT IS TEXTUAL DATA?

Textual data (or "text as data," as it is often called) usually comes in the form of *documents* as the basic units. In its simplest form, a text dataset is a collection of documents, where each document corresponds to what we would call a "case" or an "observation" in a standard social science dataset. A collection of text documents is often called a "corpus." Not surprisingly, each document in a corpus is much longer than the short strings we have seen in standard tables (e.g., country names). In many text datasets, individual documents are tagged with additional information. For example, in a corpus of political speeches, each speech can be labeled with the name of the speaker or the date it was held. Figure 12.1 illustrates what this looks like for speeches held during the United Nations General Debates, which will be introduced in more detail below.

The figure shows two speeches, one held by the US in the 34th General Debate in 1978, one by Hungary in the 52nd debate in 1997. The main data is the text of the speeches, and each of them is tagged with metadata. This means that the entire corpus of textual data can actually be represented in a tabular structure, where each of the metadata fields and the text itself correspond to a column. Figure 12.2 shows what this looks like

Country	Year	Debate	Text
USA	1978	34	We meet in this General Assembly [..]
HUN	1997	52	I am very pleased to see [..]

FIGURE 12.2. The two documents stored in a table.

for the two speeches above. Therefore, as in the chapters before, we can transform yet another type of data to a tabular format, and use many of the data operations we are already familiar with.

While we can use a *structured* data format such as a table to store an entire corpus, text data is typically considered to be *unstructured*. Here, "unstructured" refers to the main part of the data, which is the text. Unstructured means that the text does not follow a particular pattern or model, such that it is difficult to locate particular pieces of information in it. For example, it is likely that each of the texts in our example above contains information about the country holding the speech. You can see this in the speech from Hungary, which pledges the support of the "Hungarian delegation." Still, it is not straightforward to extract this information, as the country information is not explicitly flagged as such in the text, and is likely to be phrased differently in speeches of other countries. Compare this to the structured part of the dataset (the metadata). Here, we can simply look up the respective column to find out about the country holding the speech. Hence, not surprisingly, unstructured data require different and more complex methods to extract information compared to structured data such as tables.

How do we store text data digitally? As for spatial data, there exist numerous options and file formats. A first distinction we have to make is whether we store an entire corpus with different documents in a single file or as a collection of files. For the latter, we can simply use one text file for each document (see Chapter 4 for some basics about text files). When following this approach, each text file contains only text, not a CSV-encoded tabular structure. This means that there are some potential issues. For example, recall what we discussed about text encoding in Chapter 4. If text contains special characters that exist for many languages worldwide, we have to make sure that we choose an appropriate encoding. Also, we have to decide where to store document metadata if the text files themselves contain only plain text. As we will see below, this is often done by encoding metadata in file names or folder names.

A different approach for storing a document collection is to keep all documents in a single file. Again, there are many different ways for doing

this. One that you are already familiar with is the CSV format, where each document and its associated metadata corresponds to a single line. When using CSV for collections of long texts, we have to be particularly careful about how to deal with commas and line breaks in the texts. Since these characters have a particular function in CSV files (they separate fields and lines), we have to make sure that the texts containing them are properly encoded in the CSV file, for example, by enclosing them in double quotes (see Chapter 4).

12.2 Application: References to (In)equality in UN Speeches

In 2015, the United Nations adopted the *2030 Agenda for Sustainable Development*, a plan for improving economic, social, and environmental conditions worldwide. At the core of this agenda is a set of 17 *Sustainable Development Goals* (SDGs) that should be achieved by the year 2030. Goal No. 10 is reduction of inequality, both at a global scale between countries and also between different groups within countries. More details about SDG No. 10 are provided on the UN website at https://www.un.org/sustainabledevelopment/inequality/. Inequality and its reduction have not always been a high priority for the UN. For example, during the Cold War era, much of UN politics was about international security and the avoidance of violent conflict.

How can we trace the salience of different topics in the UN over time? How can we find out whether and when (in)equality became an issue of concern for deliberations at the UN? This is the kind of question that can be analyzed using statements from political actors. In our application for this chapter, we focus on speeches by UN member states at the UN General Debate, held once a year at the beginning of each session of the General Assembly at the UN Headquarters in New York. At the General Debate, each state is usually represented by its head of government. The first General Debate took place in 1946, the 2020 General Debate was held in September 2020 and commenced the 75th session of the General Assembly.

General debates last for several days and consist of a series of speeches by the representatives of the member states. In this chapter, we rely on the UN General Debate Speech Corpus (UNGDC, Baturo et al., 2017), a collection of the General Debate speeches between 1970 and 2018. As Baturo et al. (2017) note, the speeches are used by the different governments to comment on particular events and developments in the past year,

but also to emphasize pressing issues in world politics. Therefore, the collection provides us with an interesting opportunity to find out when and how inequality was mentioned in these speeches over time.

Rather than placing all the speeches into a single file, the UNGDC is distributed as a compressed archive, where each speech is stored in a single plain text file. Speeches are stored in separate directories, one for each year. The names of the text files contain information about the country holding the speech, the session (starting with 25, which corresponds to the 1970 General Debate) and the year. This is what the data structure looks like for the 25th debate in 1970:

```
Session 25 - 1970
├── ALB_25_1970.txt
├── ARG_25_1970.txt
├── AUS_25_1970.txt
└── ...
```

The dataset uses ISO three-letter codes to denote countries. In each file name, the different metadata fields are separated with an underscore. For the purpose of illustration, and to limit the computational complexity of the code examples in this chapter, we focus only on speeches by the US as one of the dominant countries in the UN. Our simple task in this chapter is to locate mentions of terms related to (in)equality in the US speeches over time. In line with the previous chapters, we do this first in R only, and later also in PostgreSQL.

12.3 WORKING WITH STRINGS IN (BASE) R

As you know, R data frames have columns with different types, one of which is character vectors for text. The character sequences (strings) we have used so far are short, such as country or party names. The texts below are much longer, but in principle can be treated exactly as the short strings we encountered so far. As mentioned above, a collection of texts can stored in a tabular file format such as CSV, in which case you can import them to R as any other CSV file. If, however, the texts are stored as separate files, this becomes a bit more difficult. This is the case for the UNGDC that we use in this chapter, where each file contains only the text of a speech, and the metadata is encoded in the file name. Luckily, there is a useful package called `readtext` that makes the import of these file collections easy. `readtext` is a companion package to the powerful

quanteda text analysis framework, which we take a closer look at below. With readtext installed, we load the package:

```
library(readtext)
```

The data repository for this chapter contains the UN General Debate speeches for the US, with one speech per file. The readtext() function is designed to import collections of files. Rather than just specifying a single file to be read, you can use the wildcard character * to specify a *pattern* of directories and files that the function should use. In our case, this patterns consists of the folder in which the text files are located (ch12), and the file name pattern *.txt. The function will then process all files ending in *.txt in the given directory. In addition, the function needs information about where the document metadata are stored. In quanteda terminology, these metadata are "document variables" or "docvars." In our case, the country, the session, and the year of the respective speech are part of the file name, which is why we set the docvarsfrom parameter accordingly. Finally, we need to specify what metadata fields are encoded in the file name. If you omit this parameter, readtext() will assign standard names for these variables. Our speech files are named such that the different document variables are separated with an underscore, which the function recognizes by default. A different separator can be set with the dvsep parameter.

```
docs <- readtext("ch12/*.txt",
   docvarsfrom = "filenames",
   docvarnames = c("country", "session", "year"))
```

The readtext() function can process many more types of text files, including PDF or MS Word. Also, it can handle different ways of storing metadata, for example, in CSV format. Let us take a closer look at what the function does. If the import is successful, the function returns an (amended) data frame, where each speech corresponds to one row (49 in total). We can use standard R syntax to output a single document, such as this one:

```
docs[1,]
```

```
readtext object consisting of 1 document and 3 docvars.
# Description: df [1 x 5]
  doc_id          text                country session  year
* <chr>           <chr>               <chr>     <int> <int>
1 USA_25_1970.txt "\"1.\t It is \"..." USA          25  1970
```

This document is the speech given by the US in 1970, at the beginning of the UN's 25th session. Each document has a unique `doc_id`, generated from the file name. The metadata (country, session, and year) are contained in the respective columns, exactly as we specified above. The most important column is `text`, which contains the text of each speech. This is a standard character variable, and we can use R's basic functions to work with it. Before we turn to our research question and study how inequality is referenced in the speeches over time, let us examine the texts in more detail. Take a closer look at the first speech by outputting the beginning of the first two paragraphs with the `substr()` function that returns a subset of a string between the given positions:

```
substr(docs[1,]$text, 1, 22)

[1] "1.\t It is my privilege"

substr(docs[1,]$text, 957, 970)

[1] "2.\tDuring this"
```

It seems that each paragraph in this speech is numbered, followed by a tab character (`\t`). Recall that the tab is one of the invisible characters we discussed in Chapter 4. If you open the corresponding file `USA_25_1970.txt` in RStudio's text editor, you can verify that the numbering of paragraphs continues in the same fashion (digits, followed by a dot, followed by a tab character). These numbers may be problematic, since they are not part of the actual speech, but also are not used consistently throughout the dataset. For example, the speech from 1996 no longer has numbered paragraphs. Therefore, it is best to clean up the texts by removing the paragraph numbers. How can we do this?

Manually searching for (and replacing) individual numbers such as "1.", "2.", etc. is not an option, as there are dozens of numbered paragraphs in some speeches. Also, it would violate one of our core rules for data processing, which is that data manipulations should be transparent and replicable, and therefore be defined in code. For these reasons, we need a better search method, where we can flexibly define a search pattern. This is what so-called *regular expressions* (in short, regex) allow us to do. Regular expressions are extremely powerful and not just limited to R; in fact, they constitute a standard feature of many other programming languages as well. Relational databases can handle regular expressions too, as we will see below.

We start by first developing a pattern to locate the paragraph numbers, and later use this pattern to eliminate them from the speeches. Before we

use the real speeches, I demonstrate the use of regular expressions using some toy examples. In R, the most important functions to be used together with regular expressions are `grep()` and the closely related `grepl()`. They require two parameters: A regex pattern and a vector of strings in which to locate the pattern. `grep()` then returns the index for each string in which it was able to locate the pattern. For simplicity, we use `grepl()`, which returns a vector of the same size as the input vector, where each entry indicates whether the search pattern occurs in the respective input string. Let us try this with a simple example. A regex pattern can be a single character, for example, the character `a`:

```
grepl("a", c("data", "management", "book", "2022."))
```

```
[1]  TRUE  TRUE FALSE FALSE
```

The character `a` occurs somewhere in the first two input strings, but not in the last two. If we refine our pattern such that it looks for the sequence `ag`, we get only one match, since this pattern only occurs in the string `management`:

```
grepl("ag", c("data", "management", "book", "2022."))
```

```
[1] FALSE  TRUE FALSE FALSE
```

Rather than particular characters, you can also search for *classes* of characters, such as all lowercase letters, or all digits. Let us try the latter. Before we do this, we need to briefly look at how R deals with strings and special characters within them, since this can interfere with how some regexp patterns are defined. Some characters in R have a special meaning. For example, as you recall from Chapter 4, a line break is denoted by `\n`, which uses the special character `\`. Another example is single (`'`) or double quotes (`"`), which are used to denote the beginning and the end of a string, and therefore cannot occur within the string itself unless we remove their special meaning. To do this, you need to "escape" them with a backslash `\`. For example, to generate an actual backslash in a string, you write `\\`. To try this, you can use the `writeLines()` function in R to output the *real* content of a string, not how it is represented in R. For example, `writeLines("\\\"")` generates the output `\"`.

Similar problems arise if we use a notation with a backslash (or other special characters) to define search patterns in regular expressions. In a regex pattern, the shortcut `\d` denotes a single digit (between 0 and 9). Since we need to escape the backslash, `\d` now becomes `\\d`. The added

backslash tells R to treat the next character as is, and not as one with a special meaning. Let us use this to locate digits in our toy example:

```
grepl("\\d", c("data", "management", "book", "2022."))
```

```
[1] FALSE FALSE FALSE  TRUE
```

Oftentimes, we want to detect *repetitions* of particular patterns, for example, a sequence of exactly four digits to search for years. This can be done with a regex *quantifier*, which indicates that a particular pattern must occur at least (or at most) a certain number of times. In its most generic form, this is denoted with curly brackets around the exact number of occurrences we want. The following example demonstrates this and we correctly locate the four-digit number in the last string:

```
grepl("\\d{4}", c("data", "management", "book", "2022."))
```

```
[1] FALSE FALSE FALSE  TRUE
```

There are different variations of the notation. For example, `{n,m}` matches the preceding patterns at least `n`, but at most `m`, times. If you would like a pattern to be present at least once (but possibly more than that), you can also use the `+` operator. Searching for at least one digit then simply becomes `d+`. Let us use this to search for a sequence of digits followed by a dot. The latter is again a special character and needs to be escaped to be interpreted as a full stop (dot).

```
grepl("\\d+\\.", c("data", "management", "book", "2022."))
```

```
[1] FALSE FALSE FALSE  TRUE
```

We are now already very close to a regex pattern that allows us to clean up the UN speeches. Recall that we need to search for a sequence of digits (at least one), followed by a dot, followed by a tab character. The latter is a literal character in a regular expression and does not need to be escaped, so we simply amend our above pattern with a `\t` before we can apply it to the speeches. To verify whether this works, we use the `grep()` function that returns the indexes of those texts where it was able to locate the pattern.

```
grep("\\d+\\.\t", docs$text)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

As we already suspected above, the numbering of paragraphs is not consistently applied in all speeches. Rather, the output of the `grep()` function shows that this pattern is only found in the first 15 speeches, which means that it stops after 1984. The `gsub()` function helps us remove these paragraph numbers. It takes as input a regex pattern (which we already have), a replacement text (which in our case is an empty string `""`, because we simply delete the paragraph numbers), and a vector of strings that should be modified. The latter is our collection of speeches, and we write the result back to our existing data frame:

```
docs$text <- gsub("\\d+\\.\t", "", docs$text)
```

This example gave you an idea of how to work with regular expressions for searching and manipulating texts. Regular expressions are a widespread and extremely powerful technique to process strings, and we have only scratched the surface of the functionality and flexibility they offer. If you want to learn more about them, you will find many useful tutorials online or in more comprehensive R introductions. As mentioned above, regex are not a feature specific to R. While the general ideas apply also to other programming languages and regex implementations, there are different dialects with slight differences in the syntax. Even though we covered regular expressions for our work with political texts, their use is by no means limited to human text and language. In fact, regular expressions can be very useful to fix issues in data files, for example, in malformed CSV files.

12.4 Natural Language Processing with quanteda

Base R and regular expressions can help you get a lot of tasks done when it comes to the management and processing of text data, as we have seen above. These tools are not specifically designed for the processing of natural language – you can use them for any types of strings. However, social science applications of text analysis mostly deal with text produced by humans, which is why we need extension libraries with features designed specifically for the processing of natural language. With the growing popularity of these approaches, a variety of software tools are now available for text analysis, including several ones for R.

Natural language processing (NLP) can be done at very different levels of sophistication. Simple approaches such as the one we use in this chapter are based on frequencies of words. More complex ones explore relationships between words, for example, by grouping them together

such that documents can be assigned to the topic(s) discussed in them. Even more sophisticated techniques for the processing of texts aim to get at the semantics of texts and their constituent parts. This usually involves the parsing of sentences to detect different classes of words such as nouns or verbs, or the extraction of subject and object in a sentence. All these more advanced methods are beyond the scope of this book, but they require the same type of input that we are dealing with in this chapter.

Almost all text analysis methods require some basic processing steps. In this section, we perform these steps with the `quanteda` library, one of the most advanced text analysis packages for R. In line with the scope of the book, however, we do not explore `quanteda`'s analysis features in depth. If you would like to learn more about the package, the online guide at https://quanteda.io/articles/pkgdown/quickstart.html is a good place to start.

Basic processing of text data involves a number of clean-up steps. One of the first is the splitting of texts into tokens, which usually correspond to words or word stems. This may seem straightforward: In English, and many other languages, words are separated by white spaces, so we can use these to separate words. However, in addition we need to deal with punctuation, which means that full stops, commas, or quotes also need to be taken into account. To do this, the computer needs instructions for what sequences of characters to assign to the same word, and when to start a new word. `quanteda` can deal with all this, so let us take a look. Once you have installed the package, you can load it with:

```
library(quanteda)
```

Since `quanteda` and `readtext` come from the same developers and are designed to work together, we can easily create a corpus from the documents imported above, from which we already removed the paragraph numbers:

```
speech_corpus <- corpus(docs)
```

Once the text is converted to a corpus, `quanteda` can easily split the texts into words and sentences. Take a look at the output of `summary(speech_corpus)` to find out how the length of the speeches (measured as the number of words or the number of sentences) varies in the speeches over time. For our application, however, we can get a first view by looking at the words we are interested in, and the context in which they occur with the `kwic()` function ("keywords in context"). We first use the `tokens()`

Published online by Cambridge University Press

function to split the text into its constituent parts, and pass these tokens on to the `kwic()` function (the output is restricted to the first three occurrences for presentational purposes). With the `pattern` parameter, we define the term(s) we are interested in. Here, it is possible to specify a single term that will be matched exactly. In our example, we use the wildcard character `*`, which matches zero or more characters. This means that we can search for "equality" but also "inequality" with this simple pattern. Note that the pattern is matched against the *individual tokens* in the text which were generated when creating the corpus. The `window` parameter specifies the context that should be displayed with each match of the pattern – in our example, we show the two tokens left and right:

```
kwic(tokens(speech_corpus), pattern = "*equality", window = 2)[1:3,]
```

```
Keyword-in-context with 3 matches.
 [USA_25_1970.txt, 1287] and human | equality | ; fifth
 [USA_25_1970.txt, 3336] of racial | equality | . The
 [USA_25_1970.txt, 3484]  justice, | equality | and self
```

The example shows you what context our keywords occur in, and what the tokens in the text look like. Try removing the restriction to the first three lines to display the entire output, and you will see that we are capturing the right words that we are interested in. "Equality" and "inequality" occur together with references to justice or race. Oftentimes, these references are made with political goals ("combating inequality"). You can try to increase the window size in the above example to see more words before or after the target terms. The output also shows that the tokenization in `quanteda` does not remove anything from the text by default – punctuation characters such as `;` or `.` are included as individual tokens.

In the next steps, we create a data structure that is very common in text analysis: a document-feature matrix (DFM). This matrix has a column for each token (*feature*) in our corpus. Each row corresponds to a *document* in our corpus, and it contains the number of times that a token occurs in that text. Recall that we have a number of rather useless tokens in our corpus. This is why we first remove punctuation and numbers from our tokens before passing them on to the `dfm()` function. Later, we also remove so-called stopwords (words such as "a," "the," etc) from our DFM. For these stopwords, it is necessary to select the language (`"en"`), since stopwords are obviously specific to each language:

```
speech_dfm <- dfm(
   tokens(speech_corpus, remove_punct = T, remove_numbers = T))
speech_dfm <- dfm_remove(speech_dfm, pattern = stopwords("en"))
```

As you can find out with `featnames(speech_dfm)`, by default the DFM converts all tokens to lowercase, otherwise the different spellings of the same word (as in "Products" and "products") would be counted as two different words. A DFM has usually lots of columns (10059, in our case), and many entries are zero because the corresponding terms do not occur in the respective document. If you print the DFM for our corpus, you can see this in the output (not shown here):

```
print(speech_dfm)
```

`quanteda` has a number of useful functions for exploring DFMs. One of them displays the most frequent words in the corpus, the "top features." Not surprisingly, for our corpus of UN speeches, the top two words are "united" and "nations" (output again restricted to the first three for presentational purposes):

```
topfeatures(speech_dfm)[1:3]
```

```
 united nations   world
   1604    1577     979
```

The DFM already contains the information we need for our applied example in this chapter. Recall that we want to count the mentions of (in)equality in the different speeches given by the US at the UN General Debate over the years. Our DFM gives us the number of times that *any* term in the corpus appears in the respective speech. Therefore, all we need to do is select the relevant terms from our DFM and extract the counts. We achieve this by creating a new, restricted DFM that only contains terms related to inequality. The `dfm_select()` function does this for us, and it accepts the same type of filter pattern as the `kwic()` function above.

```
dfm_ineq <- dfm_select(speech_dfm, pattern = "*equality")
```

By default, the function keeps the specified terms, which is exactly what we want. All that is left for us to do is to compute the row sums of the reduced DFM `dfm_ineq`, which gives us the number of times that either "inequality" or "equality" is mentioned in the speech. We add this count as an additional variable to our corpus:

```
speech_corpus$ineq_count <- rowSums(dfm_ineq)
```

At the end of the chapter, we create a simple plot using the `year` field and the newly created `ineq_count` field from the corpus.

12.5 Using PostgreSQL to Manage Documents

In the final part of this chapter, we will show how to use a relational database to store and manage text documents. As we saw in previous chapters, the main focus of databases is the storage and efficient retrieval of data, not the analysis. This is why PostgreSQL is very limited when it comes to the processing of natural language; if your workflow involves text data stored in a database, you will typically export it for further processing in a specialized package such as `quanteda`. Nevertheless, it is useful to take a brief look at how PostgreSQL deals with text data, and how you can query the data with natural language searches. As in the previous chapters, we assume that you use a new database for this chapter, called `textdata`. We connect to our database with

```
library(RPostgres)
db <- dbConnect(Postgres(),
   dbname = "textdata",
   user = "postgres",
   password = "pgpasswd")
```

and import the UN speeches into a new table `speeches`. As you know from above, `readtext()` returns an extended data frame. We cannot directly send it to the database, which is why we convert it to a real data frame beforehand:

```
docs <- readtext("ch12/*.txt",
   docvarsfrom = "filenames",
   docvarnames = c("country", "session", "year"))
dbWriteTable(db, "speeches", as.data.frame(docs))
```

The table we created has five fields: the `doc_id` (the file name of the corresponding text file); the three docvars `country`, `session`, and `year`; and the `text` column that contains the text of the speech. As a next step, we again remove the line numbering from the speeches. Above, we have seen how to do this with regular expressions. Regex are not a feature specific to R; they also exist for PostgreSQL with a notation similar to the one described above. This is why we can use the search pattern `\d+\.\t` – which matches one or more digits followed by a dot, followed by a tab

character – also in our database. Let us first use the regexp search operator, the tilde `~`. This operator performs pattern matching. In the following query, we count the number of documents from our collection where the `text` field matches our search pattern (i.e., contains it at least once). Note that we again have to escape the backslashes properly, so that R passes them correctly to the database:

```
dbGetQuery(db, "SELECT count(*) FROM speeches WHERE text ~ '\\d+\\.\t'")
```

```
  count
1    15
```

The result is the same as above: 15 documents match our search pattern, which are the first 15 speeches in the dataset. How can we clean up the texts of these speeches? For this, we use the `regexp_replace()` function in PostgreSQL. It takes a string, a search pattern, and a replacement string, and returns a new string in which all occurrences of the search pattern have been replaced with the replacement string. The pattern we need is the same as above, and our replacement string is the empty string `''`, since we only want to delete the line number. In addition, the function takes optional control flags. Here, we must set the `g` (global) flag to extend the search/replace to *all* instances of the pattern, and not just the first one. There is no danger in applying this function to *all* texts in our sample; since the pattern is only found in the first 15 speeches, the others will remain unaffected:

```
dbExecute(db,
   "UPDATE speeches
   SET text=regexp_replace(text, '\\d+\\.\t', '', 'g')")
```

Having done some basic clean-up, we can now proceed to explore how to select documents from our database according to particular words in the text. One way to do this is the `~` operator, which allows us to specify a regex search pattern. Oftentimes, however, we do not really need regular expressions, which is why there is a simpler way to search text fields: the `LIKE` operator. Here, the syntax for the search pattern is much simpler. `LIKE` expects normal strings, which can contain two types of placeholders: the underscore (which matches any single character) and the percentage sign (which matches an arbitrary sequence of characters). How does this work in practice? First, we try to use `LIKE` with a search string that does not have any placeholders:

```
dbGetQuery(db, "SELECT count(*) FROM speeches WHERE text LIKE 'equality'")
```

```
  count
1     0
```

This query does not find any matching documents – why? The reason is that the search pattern `equality` is matched against the entire `text` column, so a match would only occur for any document where the *entire text* of the speech is "equality." Of course, there is no single speech with this content in our collection. This is why we need the `%` placeholder, which matches an arbitrary sequence of characters. In this query

```
dbGetQuery(db,
   "SELECT count(*) FROM speeches
   WHERE text LIKE '%equality%'")

  count
1    11
```

we allow an arbitrary number of characters to occur before and after "equality." This matches any speech where "equality" occurs at least once in the text, which is the case for eleven of them. Similar to `grep()` and its related functions in R, string matching operators such as `LIKE` are designed to work with strings in general. If you use them with natural language, they have no knowledge of what a word is, or that particular characters such as the dot can have a special meaning in language. This is why we need special extensions. PostgreSQL has a built-in set of functions for "full text search," which help us process natural language. However, as the name suggests, it is designed primarily for searching natural language documents, so its applicability is much more limited compared to packages such as `quanteda`.

Recall that the first step in dealing with natural language is usually the clean-up of the text: We identify the tokens in the text and remove stopwords and punctuation. PostgreSQL does this by converting a document to a text search vector (`tsvector`), that contains a reduced form of the text. Similar to our document-feature matrix above, this vector contains the list of tokens in the text as well as the positions where they occur in the text. Therefore, once processed in this way, it is much easier to search for natural language terms in the text, since the DBMS only needs to go through this vector rather than the entire text. The creation of this vector is done with the `to_tsvector()` function, which converts a given string to a text search vector. Let us try this first with a simple example before applying it to the UN speeches:

```
dbGetQuery(db,
   "SELECT to_tsvector('english', 'The problems the world faces today')")

                                to_tsvector
1 'face':5 'problem':2 'today':6 'world':4
```

The output shows you what a vector looks like. It contains four tokens, along with the positions of these tokens in the text. For example, `world` is the fourth token in the text. There are a few things to note here. Most importantly, tokenization is language specific, so we need to specify `english` as the language. Now, take a look at the second token in the text, `problems`. This token is included as `problem` in the vector, with the "s" removed. The reason is that text search vectors trim the words to their word *stems*, so `problems` and `problem` are reduced to the same stem. Also, stopwords such as `the` have been removed from the index, and all the tokens are lower case.

The creation of text search vectors is computationally costly, so it is good practice to compute them once and save them in a separate column, such that you can use them later when searching the documents. The following code creates a new column of type `tsvector`, computes the vectors, and indexes the column using a special type of index (`gin`) to speed up data retrieval (see Chapter 10):

```
dbExecute(db, "ALTER TABLE speeches ADD COLUMN tokens tsvector")
dbExecute(db, "UPDATE speeches SET tokens = to_tsvector('english', text)")
dbExecute(db, "CREATE INDEX ON speeches USING gin(tokens)")
```

How do we use the text search vectors in practice? In PostgreSQL, we can now run a text search *query* against the vectors we created. A text search query uses a very simple syntax, similar to web search engines. In its simplest form, such a query is just a single word, but you can also connect different words with logical AND (`&`) and OR operators (`|`). We create a query with the `to_tsquery()` function. Importantly, PostgreSQL internally reduces the query in the same way as a text search vector. The advantage is that we do not have to worry about the different forms of a word – for example, `inequality` and `inequalities` are internally reduced to the same form:

```
dbGetQuery(db,
   "SELECT
      to_tsquery('english', 'inequality'),
      to_tsquery('english', 'inequalities')")

  to_tsquery to_tsquery..2
1    'inequ'       'inequ'
```

Now let us apply a simple query to our documents. For text searches, there is a special operator, `@@`, which determines whether a given text search vector matches a text search query. Here, we count the number of documents that match the query `inequality`:

```
dbGetQuery(db,
   "SELECT doc_id FROM speeches
   WHERE tokens @@ to_tsquery('english', 'inequality')")
```

```
           doc_id
1 USA_33_1978.txt
2 USA_53_1998.txt
3 USA_70_2015.txt
4 USA_71_2016.txt
```

If you go through the four speeches that match our query (the speeches for 1978, 1998, 2015, and 2016), you will see that not all of them contain the word `inequality` or some other form of it. The 1978 speech only talks about "international inequities and poverty" – however, since the text search applies stemming to the tokens, `inequality` and `inequities` are reduced to the same stem `inequ`, which is why we get a match also for the 1978 speech. This is exactly what we want in this case, since the 1978 speech talks about inequality between countries, which is what we are interested in. However, in other cases, the results of text searches can be too inclusive. Consider the following example, where we amend our pattern such that it searches of `inequality` or `equality`. Now, we get 34 matching documents:

```
dbGetQuery(db,
   "SELECT count(*) AS num_ineq FROM speeches
   WHERE tokens @@ to_tsquery('english', 'inequality | equality')")
```

```
  num_ineq
1       34
```

The reason is that `equality` is reduced to `equal`, which occurs frequently without any connection to inequality, for example, in sentences such as "Equally important, we hope that [..]." Hence, while text search can be a powerful and flexible tool to explore natural language, you have to be aware of the uncertainties when doing so. Before we proceed to show the result of our applied example, we close the database connection properly:

```
dbDisconnect(db)
```

12.6 Results: References to (In)equality in UN Speeches

The reduction of inequality has been one of the UN's *Sustainable Development Goals*, and the purpose of our exercise is to track the use of this

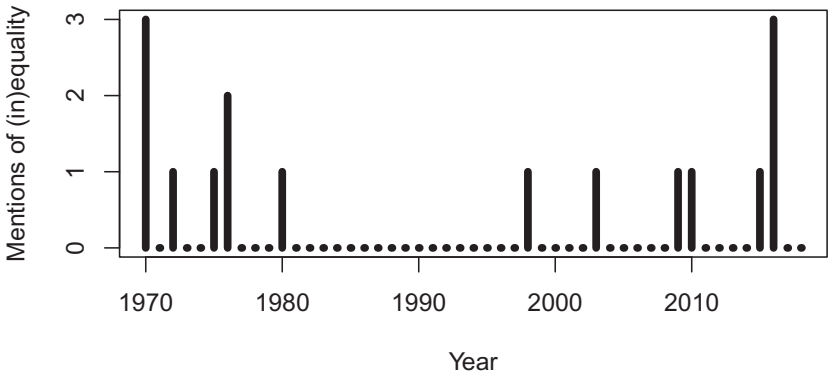

FIGURE 12.3. Number of mentions of (in)equality in UN General Debate speeches by the US over time.

and related terms over the years. We now use the statistics we computed with the help of the `quanteda` package. Above, we extracted the number of times that `inequality` or `equality` appear in the speeches from the DFM. In the simple plot in Figure 12.3, we use the two fields `speech_corpus$year` and `speech_corpus$ineq_count` for plotting.

We can see that in the early years of the sample, the US made several references to (in)equality in the UN General Debate speeches. This may be partly due to the civil rights movement in the US and political attention it had triggered to issues of inequality. The 1970s, however, were followed by a period without any mention in the 1990s, before the term came up more often again during the 2000s and later. The peak at the end of the study period (2016) coincides roughly with the adoption of the SDGs in 2015. Thus, as we can see based on this simple example, there is considerable variation in the salience of inequality in international politics over time, although it is not very prominent throughout.

12.7 SUMMARY AND OUTLOOK

Much research in the social sciences now relies on natural language data. In this chapter, we covered text as data, and how it can be processed in R and in PostgreSQL. Even though documents and their metadata can be stored in a tabular (structured) format, the texts themselves are examples of "unstructured" data. That is, *within* a text, we usually have no explicit structure, and the format and content of texts varies between documents. In this chapter, we used two different approaches to process text data. First, you can treat text simply as long strings, and use standard

string functions. Among these, we have introduced regular expressions, a very powerful method to search strings for particular patterns and replace them. Regular expressions are available in R and in PostgreSQL, but also in almost any other programming language. While useful when processing text data, they can be of great help for other tasks with strings, for example, when fixing malformed data files.

The second, more sophisticated approach we have illustrated in this chapter is to treat texts as natural language, and apply specialized methods for this. For R, the `quanteda` package is a good choice, but there are also other options such as `tidytext`, which integrates nicely into the `tidyverse` universe. These packages can perform a variety of NLP tasks, such as the splitting of a text into tokens, the elimination of stopwords and punctuation, or the reduction of words to their stems. For all of this, specialized knowledge of the particular features of a language are required, for example, how different sentences are separated, or how words can be trimmed to their stem. The relational database PostgreSQL has some basic functionality for doing this, which can come in handy if you keep a collection of documents on a centralized server and need to do fast and flexible lookups. For more advanced analysis of text, however, it is usually required to export the documents to R and use a text analysis package such as `quanteda`, whose functionality is much more advanced. For your future work with text data, here are some useful pointers:

- *Practice the use of regular expressions:* We began this chapter with an introduction to some standard string operations, which are available both in R and PostgreSQL. Among these, regular expressions are particularly powerful, but at the same time remain challenging even for experienced programmers. If you plan to work more with text data in the future, I recommend that you practice the use of regular expressions, since there are many operators and shortcuts we did not discuss in this chapter.
- *Compression is highly effective for text data:* Remember that we discussed the use of file compression in Chapter 4? This is particularly important if you store text datasets in files, since they can became very large. File compression works well with text files, and you can reduce the required disk space considerably for text data projects with tools such as `zip` or `gzip`. It is up to you to implement compression in a file-based workflow; PostgreSQL enables it automatically for text data.
- *Large files can be loaded directly into PostgreSQL:* So far, we have used R's DBI functions (such as `dbWriteTable()`) to import data into our

database. For very large CSV files, there is another way to this, which bypasses R completely. The PostgreSQL server can read files on your disk directly, which is much faster – for details, see the PostgreSQL documentation at https://www.postgresql.org/docs/current/sql-copy.html. There are some restrictions, however: This feature can only process tabular data in CSV (or similar) formats, and it becomes more difficult to use if you connect to PostgreSQL running on a remote server.

- *What if you need more flexible fuzzy string matching?* In the chapter, we discussed a number of ways in which you can specify search patterns to be located in strings. These approaches allow you to use different wildcard characters. Another way to implement non-precise, fuzzy string matching is by means of the "Levenshtein distance," which is the number of characters that need to be changed when transforming one string into another. This is a common measure of similarity between strings, and you can use it in R with the `adist()` and the `agrep()`/`agrepl()` functions. In PostgreSQL, you can use the `levenshtein()` function, which is part of the `fuzzystrmatch` extension.