Preface

"DH has long used advanced computational techniques and ML to aid humanistic enquiry. This includes predictive analytics, which uses ML techniques as well as traditional statistical methods. It can be used to predict the past using past data."

Modeling as a statistical practice can encompass a wide variety of activities. This book focuses on *supervised or predictive modeling for text*, using text data to make predictions about the world around us. We use the tidymodels[1] framework for modeling, a consistent and flexible collection of R packages developed to encourage good statistical practice.

Supervised machine learning using text data involves building a statistical model to estimate some output from input that includes language. The two types of models we train in this book are regression and classification. Think of regression models as predicting numeric or continuous outputs, such as predicting the year of a United States Supreme Court opinion from the text of that opinion. Think of classification models as predicting outputs that are discrete quantities or class labels, such as predicting whether a GitHub issue is about documentation or not from the text of the issue. Models like these can be used to make predictions for new observations, to understand what features or characteristics contribute to differences in the output, and more. We can evaluate our models using performance metrics to determine which are best, which are acceptable for our specific context, and even which are fair.

Text data is important for many domains, from healthcare to marketing to the digital humanities, but specialized approaches are necessary to create features (predictors) for machine learning from language.

Natural language that we as speakers and/or writers use must be dramatically transformed to a machine-readable, numeric representation to be ready for computation. In this book, we explore typical text preprocessing steps from the ground up and consider the effects of these steps. We also show how to fluently use the **textrecipes** R package (Hvitfeldt 2020a) to prepare text data within a modeling pipeline.

[1] https://www.tidymodels.org/

Silge and Robinson (2017) provides a practical introduction to text mining with R using tidy data principles, based on the **tidytext** package. If you have already started on the path of gaining insight from your text data, a next step is using that text directly in predictive modeling. Text data contains within it latent information that can be used for insight, understanding, and better decision-making, and predictive modeling with text can bring that information and insight to light. If you have already explored how to analyze text as demonstrated in Silge and Robinson (2017), this book will move one step further to show you how to *learn and make predictions* from that text data with supervised models. If you are unfamiliar with this previous work, this book will still provide a robust introduction to how text can be represented in useful ways for modeling and a diverse set of supervised modeling approaches for text.

see the present then see outside of it

Outline

The book is divided into three sections. We make a (perhaps arbitrary) distinction between *machine learning methods* and *deep learning methods* by defining deep learning as any kind of multilayer neural network (LSTM, bi-LSTM, CNN) and machine learning as anything else (regularized regression, naive Bayes, SVM, random forest). We make this distinction both because these different methods use separate software packages and modeling infrastructure, and from a pragmatic point of view, it is helpful to split up the chapters this way.

remember this definition from Dan Ryans CT

- **Natural language features:** How do we transform text data into a representation useful for modeling? In these chapters, we explore the most common preprocessing steps for text, when they are helpful, and when they are not.

- **Machine learning methods:** We investigate the power of some of the simpler and more lightweight models in our toolbox.

- **Deep learning methods:** Given more time and resources, we see what is possible once we turn to neural networks.

Some of the topics in the second and third sections overlap as they provide different approaches to the same tasks.

Throughout the book, we will demonstrate with examples and build models using a selection of text data sets. A description of these data sets can be found in Appendix B.

We use three kinds of info boxes throughout the book to invite attention to notes and other ideas.

Some boxes call out warnings or possible problems to watch out for.

Boxes marked with hexagons highlight information about specific R packages and how they are used. We use **bold** for the names of R packages.

Topics this book will not cover

This book serves as a thorough introduction to prediction and modeling with text, along with detailed practical examples, but there are many areas of natural language processing we do not cover. The *CRAN Task View on Natural Language Processing*[2] provides details on other ways to use R for computational linguistics. Specific topics we do not cover include:

- **Reading text data into memory:** Text data may come to a data practitioner in any of a long list of heterogeneous formats. Text data exists in PDFs, databases, plain text files (single or multiple for a given project), websites, APIs, literal paper, and more. The skills needed to access and sometimes wrangle text data sets so that they are in memory and ready for analysis are so varied and extensive that we cannot hope to cover them in this book. We point readers to R packages such as **readr** (Wickham and Hester 2020), **pdftools** (Ooms 2020a), and **httr** (Wickham 2020), which we have found helpful in these tasks.

[2] https://cran.r-project.org/web/views/NaturalLanguageProcessing.html

- **Unsupervised machine learning for text:** Silge and Robinson (2017) provide an introduction to one method of unsupervised text modeling, and Chapter 5 does dive deep into word embeddings, which learn from the latent structure in text data. However, many more unsupervised machine learning algorithms can be used for the goal of learning about the structure or distribution of text data when there are no outcome or output variables to predict.

- **Text generation:** The deep learning model architectures we discuss in Chapters 8, 9, and 10 can be used to generate new text, as well as to model existing text. Chollet and Allaire (2018) provide details on how to use neural network architectures and training data for text generation.

- **Speech processing:** Models that detect words in audio recordings of speech are typically based on many of the principles outlined in this book, but the training data is *audio* rather than written text. R users can access pre-trained speech-to-text models via large cloud providers, such as Google Cloud's Speech-to-Text API accessible in R through the **googleLanguageR** package (Edmondson 2020).

- **Machine translation:** Machine translation of text between languages, based on either older statistical methods or newer neural network methods, is a complex, involved topic. Today, the most successful and well-known implementations of machine translation are proprietary, because large tech companies have access to both the right expertise and enough data in multiple languages to train successful models for general machine translation. Google is one such example, and Google Cloud's Translation API is again available in R through the **googleLanguageR** package.

Who is this book for?

This book is designed to provide practical guidance and directly applicable knowledge for data scientists and analysts who want to integrate text into their modeling pipelines.

We assume that the reader is somewhat familiar with R, predictive modeling concepts for non-text data, and the **tidyverse**[3] family of packages (Wickham et al. 2019). For users who don't have this background with tidyverse code, we recommend *R for Data Science*[4] (Wickham and Grolemund 2017). Helpful

[3] https://www.tidyverse.org/

[4] http://r4ds.had.co.nz/

resources for getting started with modeling and machine learning include a free interactive course[5] developed by one of the authors (JS) and *Hands-On Machine Learning with R*[6] (Boehmke and Greenwell 2019), as well as *An Introduction to Statistical Learning*[7] (James et al. 2013).

We don't assume an extensive background in text analysis, but *Text Mining with R*[8] (Silge and Robinson 2017), by one of the authors (JS) and David Robinson, provides helpful skills in exploratory data analysis for text that will promote successful text modeling. This book is more advanced than *Text Mining with R* and will help practitioners use their text data in ways not covered in that book.

Acknowledgments

We are so thankful for the contributions, help, and perspectives of people who have supported us in this project. There are several we would like to thank in particular.

We would like to thank Max Kuhn and Davis Vaughan for their investment in the **tidymodels** packages, David Robinson for his collaboration on the **tidytext** package, and Yihui Xie for his work on **knitr**, **bookdown**, and the R Markdown ecosystem. Thank you to Desirée De Leon for the site design of the online work and to Sarah Lin for the expert creation of the published work's index. We would also like to thank Carol Haney, Kasia Kulma, David Mimno, Kanishka Misra, and an additional anonymous technical reviewer for their detailed, insightful feedback that substantively improved this book, as well as our editor John Kimmel for his perspective and guidance during the process of writing and publishing.

This book was written in the open, and multiple people contributed via pull requests or issues. Special thanks goes to the four people who contributed via GitHub pull requests (in alphabetical order by username): @fellennert, Riva Quiroga (@rivaquiroga), Darrin Speegle (@speegled), Tanner Stauss (@tm-stauss).

Note box icons by Smashicons from flaticon.com.

[5] https://supervised-ml-course.netlify.com/

[6] https://bradleyboehmke.github.io/HOML/

[7] http://faculty.marshall.usc.edu/gareth-james/ISL/

[8] https://www.tidytextmining.com/

Colophon

This book was written in RStudio[9] using **bookdown**[10]. The website[11] is hosted via GitHub Pages[12], and the complete source is available on GitHub[13]. We generated all plots in this book using **ggplot2**[14] and its light theme (`theme_light()`). The `autoplot()` method for `conf_mat()`[15] has been modified slightly to allow colors; modified code can be found online[16].

This version of the book was built with R version 4.1.0 (2021-05-18) and the following packages:

package	version	source
bench	1.1.1	CRAN (R 4.1.0)
bookdown	0.23	CRAN (R 4.1.0)
broom	0.7.9	CRAN (R 4.1.0)
corpus	0.10.2	CRAN (R 4.1.0)
dials	0.0.9	CRAN (R 4.1.0)
discrim	0.1.1	CRAN (R 4.1.0)
doParallel	1.0.16	CRAN (R 4.1.0)
glmnet	4.1-1	CRAN (R 4.1.0)
gt	0.3.1	CRAN (R 4.1.0)
hcandersenr	0.2.0	CRAN (R 4.1.0)
htmltools	0.5.1.1	CRAN (R 4.1.0)
htmlwidgets	1.5.3	CRAN (R 4.1.0)
hunspell	3.0.1	CRAN (R 4.1.0)
irlba	2.3.3	CRAN (R 4.1.0)
jiebaR	0.11	CRAN (R 4.1.0)
jsonlite	1.7.2	CRAN (R 4.1.0)
kableExtra	1.3.4	CRAN (R 4.1.0)
keras	2.4.0	CRAN (R 4.1.0)
klaR	0.6-15	CRAN (R 4.1.0)
LiblineaR	2.10-12	CRAN (R 4.1.0)
lime	0.5.2	CRAN (R 4.1.0)
lobstr	1.1.1	CRAN (R 4.1.0)
naivebayes	0.9.7	CRAN (R 4.1.0)

[9] https://www.rstudio.com/ide/

[10] https://bookdown.org

[11] https://smltar.com

[12] https://pages.github.com

[13] https://github.com/EmilHvitfeldt/smltar

[14] https://ggplot2.tidyverse.org

[15] https://yardstick.tidymodels.org/reference/conf_mat.html

[16] https://github.com/EmilHvitfeldt/smltar/blob/master/_common.R

package	version	source
parsnip	0.1.6	CRAN (R 4.1.0)
prismatic	1.0.0	CRAN (R 4.1.0)
quanteda	3.1.0	CRAN (R 4.1.0)
ranger	0.13.1	CRAN (R 4.1.0)
recipes	0.1.16	CRAN (R 4.1.0)
remotes	2.4.0	CRAN (R 4.1.0)
reticulate	1.20	CRAN (R 4.1.0)
rsample	0.1.0	CRAN (R 4.1.0)
rsparse	0.4.0	CRAN (R 4.1.0)
scico	1.2.0	CRAN (R 4.1.0)
scotus	1.0.0	Github (EmilHvitfeldt/scotus)
servr	0.23	CRAN (R 4.1.0)
sessioninfo	1.1.1	CRAN (R 4.1.0)
slider	0.2.2	CRAN (R 4.1.0)
SnowballC	0.7.0	CRAN (R 4.1.0)
spacyr	1.2.1	CRAN (R 4.1.0)
stopwords	2.2	CRAN (R 4.1.0)
styler	1.5.1	CRAN (R 4.1.0)
text2vec	0.6	CRAN (R 4.1.0)
textdata	0.4.1	CRAN (R 4.1.0)
textfeatures	0.3.3	CRAN (R 4.1.0)
textrecipes	0.4.1	CRAN (R 4.1.0)
tfruns	1.5.0	CRAN (R 4.1.0)
themis	0.1.4	CRAN (R 4.1.0)
tidymodels	0.1.3	CRAN (R 4.1.0)
tidytext	0.3.1	CRAN (R 4.1.0)
tidyverse	1.3.1	CRAN (R 4.1.0)
tokenizers	0.2.1	CRAN (R 4.1.0)
tokenizers.bpe	0.1.0	CRAN (R 4.1.0)
tufte	0.10	CRAN (R 4.1.0)
tune	0.1.5	CRAN (R 4.1.0)
UpSetR	1.4.0	CRAN (R 4.1.0)
vip	0.3.2	CRAN (R 4.1.0)
widyr	0.1.4	CRAN (R 4.1.0)
workflows	0.2.3	CRAN (R 4.1.0)
yardstick	0.0.8	CRAN (R 4.1.0)

1

Language and modeling

Machine learning and deep learning models for text are executed by computers, but they are designed and created by human beings using language generated by human beings. As natural language processing (NLP) practitioners, we bring our assumptions about what language is and how language works into the task of creating modeling features from natural language and using those features as inputs to statistical models. This is true *even when* we don't think about how language works very deeply or when our understanding is unsophisticated or inaccurate; speaking a language is not the same as having an explicit knowledge of how that language works. We can improve our machine learning models for text by heightening that knowledge.

Throughout the course of this book, we will discuss creating predictors or features from text data, fitting statistical models to those features, and how these tasks are related to language. Data scientists involved in the everyday work of text analysis and text modeling typically don't have formal training in how language works, but there is an entire field focused on exactly that, *linguistics*.

1.1 Linguistics for text analysis

Briscoe (2013) provides helpful introductions to what linguistics is and how it intersects with the practical computational field of natural language processing. The broad field of linguistics includes subfields focusing on different aspects of language, which are somewhat hierarchical, as shown in Table 1.1.

These fields each study a different level at which language exhibits organization. When we build supervised machine learning models for text data, we use these levels of organization to create *natural language features*, i.e., predictors or inputs for our models. These features often depend on the morphological characteristics of language, such as when text is broken into sequences of characters for a recurrent neural network deep learning model. Sometimes these features depend on the syntactic characteristics of language, such as when models use part-of-speech information. These roughly hierarchical levels of

DOI: 10.1201/9781003093459-1

TABLE 1.1: Some subfields of linguistics, moving from smaller structures to broader structures

Linguistics subfield	What does it focus on?
Phonetics	Sounds that people use in language
Phonology	Systems of sounds in particular languages
Morphology	How words are formed
Syntax	How sentences are formed from words
Semantics	What sentences mean
Pragmatics	How language is used in context

organization are key to the process of transforming unstructured language to a mathematical representation that can be used in modeling.

At the same time, this organization and the rules of language can be ambiguous; our ability to create text features for machine learning is constrained by the very nature of language. Beatrice Santorini, a linguist at the University of Pennsylvania, compiles examples of linguistic ambiguity from news headlines[1]:

- Include Your Children When Baking Cookies
- March Planned For Next August
- Enraged Cow Injures Farmer with Ax
- Wives Kill Most Spouses In Chicago

If you don't have knowledge about what linguists study and what they know about language, these news headlines are just hilarious. To linguists, these are hilarious because they exhibit certain kinds of semantic ambiguity.

Notice also that the first two subfields on this list are about sounds, i.e., speech. Most linguists view speech as primary, and writing down language as text as a technological step.

Remember that some language is signed, not spoken, so the description laid out here is itself limited.

[1] https://www.ling.upenn.edu/~beatrice/humor/headlines.html

Written text is typically less creative and further from the primary language than we would wish. This points out how fundamentally limited modeling from written text is. Imagine that the abstract language data we want exists in some high-dimensional latent space; we would like to extract that information using the text somehow, but it just isn't completely possible. Any features we create or model we build are inherently limited.

1.2 A glimpse into one area: morphology

How can a deeper knowledge of how language works inform text modeling? Let's focus on **morphology**, the study of words' internal structures and how they are formed, to illustrate this. Words are medium to small in length in English; English has a moderately low ratio of morphemes (the smallest unit of language with meaning) to words while other languages like Turkish and Russian have a higher ratio of morphemes to words (Bender 2013). Related to this, languages can be either more analytic (like Mandarin or modern English, breaking up concepts into separate words) or synthetic (like Hungarian or Swahili, combining concepts into one word).

Morphology focuses on how morphemes such as prefixes, suffixes, and root words come together to form words. Some languages, like Danish, use many compound words. Danish words such as "brandbil" (fire truck), "politibil" (police car), and "lastbil" (truck) all contain the morpheme "bil" (car) and start with prefixes denoting the type of car. Because of these compound words, some nouns seem more descriptive than their English counterpart; "vaskebjørn" (raccoon) splits into the morphemes "vaske" and "bjørn," literally meaning "washing bear"[2]. When working with Danish and other languages with compound words, such as German, compound splitting to extract more information can be beneficial (Sugisaki and Tuggener 2018). However, even the very question of what a word is turns out to be difficult, and not only for languages other than English. Compound words in English like "real estate" and "dining room" represent one concept but contain whitespace.

The morphological characteristics of a text data set are deeply connected to preprocessing steps like tokenization (Chapter 2), removing stop words (Chapter 3), and even stemming (Chapter 4). These preprocessing steps for creating natural language features, in turn, can have significant effects on model predictions or interpretation.

[2]The English word "raccoon" derives from an Algonquin word meaning, "scratches with his hands!"

1.3 Different languages

We believe that most of the readers of this book are probably native English speakers, and certainly most of the text used in training machine learning models is English. However, English is by no means a dominant language globally, especially as a native or first language. As an example close to home for us, of the two authors of this book, one is a native English speaker and one is not. According to the comprehensive and detailed Ethnologue project[3], less than 20% of the world's population speaks English at all.

Bender (2011) provides guidance to computational linguists building models for text, for any language. One specific point she makes is to name the language being studied.

> **Do** state the name of the language that is being studied, even if it's English. Acknowledging that we are working on a particular language foregrounds the possibility that the techniques may in fact be language-specific. Conversely, neglecting to state that the particular data used were in, say, English, gives [a] false veneer of language-independence to the work.

This idea is simple (acknowledge that the models we build are typically language-specific) but the #BenderRule[4] has led to increased awareness of the limitations of the current state of this field. Our book is not geared toward academic NLP researchers developing new methods, but toward data scientists and analysts working with everyday data sets; this issue is relevant even for us. Name the languages used in training models (Bender 2019), and think through what that means for their generalizability. We will practice what we preach and tell you that most of the text used for modeling in this book is English, with some text in Danish and a few other languages.

[3] https://www.ethnologue.com/language/eng

[4] https://twitter.com/search?q=%23BenderRule

1.4 Other ways text can vary

The concept of differences in language is relevant for modeling beyond only the broadest language level (for example, English vs. Danish vs. German vs. Farsi). Language from a specific dialect often cannot be handled well with a model trained on data from the same language but not inclusive of that dialect. One dialect used in the United States is African American Vernacular English (AAVE). Models trained to detect toxic or hate speech are more likely to falsely identify AAVE as hate speech (Sap et al. 2019); this is deeply troubling not only because the model is less accurate than it should be, but because it amplifies harm against an already marginalized group.

Language is also changing over time. This is a known characteristic of language; if you notice the evolution of your own language, don't be depressed or angry, because it means that people are using it! Teenage girls are especially effective at language innovation and have been for centuries (McCulloch 2015); innovations spread from groups such as young women to other parts of society. This is another difference that impacts modeling.

Differences in language relevant for models also include the use of slang, and even the context or medium of that text.

Consider two bodies of text, both mostly standard written English, but one made up of tweets and one made up of medical documents. If an NLP practitioner trains a model on the data set of tweets to predict some characteristics of the text, it is very possible (in fact, likely, in our experience) that the model will perform poorly if applied to the data set of medical documents[5]. Like machine learning in general, text modeling is exquisitely sensitive to the data used for training. This is why we are somewhat skeptical of AI products such as sentiment analysis APIs, not because they *never* work well, but because they work well only when the text you need to predict from is a good match to the text such a product was trained on.

[5]Practitioners have built specialized computational resources for medical text (Johnson 1999).

1.5 Summary

Linguistics is the study of how language works, and while we don't believe real-world NLP practitioners must be experts in linguistics, learning from such domain experts can improve both the accuracy of our models and our understanding of why they do (or don't!) perform well. Predictive models for text reflect the characteristics of their training data, so differences in language over time, between dialects, and in various cultural contexts can prevent a model trained on one data set from being appropriate for application in another. A large amount of the text modeling literature focuses on English, but English is not a dominant language around the world.

1.5.1 In this chapter, you learned:

- that areas of linguistics focus on topics from sounds to how language is used
- how a topic like morphology is connected to text modeling steps
- to identify the language you are modeling, even if it is English
- about many ways language can vary and how this can impact model results

2

Tokenization

To build features for supervised machine learning from natural language, we need some way of representing raw text as numbers so we can perform computation on them. Typically, one of the first steps in this transformation from natural language to feature, or any of kind of text analysis, is *tokenization*. Knowing what tokenization and tokens are, along with the related concept of an n-gram, is important for almost any natural language processing task.

! The first step is usually to tokenize so we can do real computation!

2.1 What is a token?

In R, text is typically represented with the *character* data type, similar to strings in other languages. Let's explore text from fairy tales written by Hans Christian Andersen, available in the **hcandersenr** package (Hvitfeldt 2019a). This package stores text as lines such as those you would read in a book; this is just one way that you may find text data in the wild and does allow us to more easily read the text when doing analysis. If we look at the first paragraph of one story titled "The Fir-Tree," we find the text of the story is in a character vector: a series of letters, spaces, and punctuation stored as a vector.

! "what is a character vector?"

The **tidyverse** is a collection of packages for data manipulation, exploration, and visualization.

```
library(tokenizers)
library(tidyverse)
library(tidytext)
library(hcandersenr)
```

opening up all the packages we'll need for this project

DOI: 10.1201/9781003093459-2

```
the_fir_tree <- hcandersen_en %>%
  filter(book == "The fir tree") %>%
  pull(text)

head(the_fir_tree, 9)
```

```
#> [1] "Far down in the forest, where the warm sun and the fresh air made a
sweet"
#> [2] "resting-place, grew a pretty little fir-tree; and yet it was not happy,
it"
#> [3] "wished so much to be tall like its companions— the pines and firs which
grew"
#> [4] "around it. The sun shone, and the soft air fluttered its leaves, and
the"
#> [5] "little peasant children passed by, prattling merrily, but the fir-tree
heeded"
#> [6] "them not. Sometimes the children would bring a large basket of
raspberries or"
#> [7] "strawberries, wreathed on a straw, and seat themselves near the
fir-tree, and"
#> [8] "say, \"Is it not a pretty little tree?\" which made it feel more
unhappy than"
#> [9] "before."
```

The first nine lines stores the first paragraph of the story, each line consisting of a series of character symbols. These elements don't contain any metadata or information to tell us which characters are words and which aren't. Identifying these kinds of boundaries between words is where the process of tokenization comes in.

In tokenization, we take an input (a string) and a token type (a meaningful unit of text, such as a word) and split the input into pieces (tokens) that correspond to the type (Manning, Raghavan, and Schütze 2008). Figure 2.1 outlines this process.

Most commonly, the meaningful unit or type of token that we want to split text into units of is a **word**. However, it is difficult to clearly define what a word is, for many or even most languages. Many languages, such as Chinese, do not use white space between words at all. Even languages that do use white space, including English, often have particular examples that are ambiguous (Bender 2013). Romance languages like Italian and French use pronouns and negation words that may better be considered prefixes with a space, and English contractions like "didn't" may more accurately be considered two words with no space.

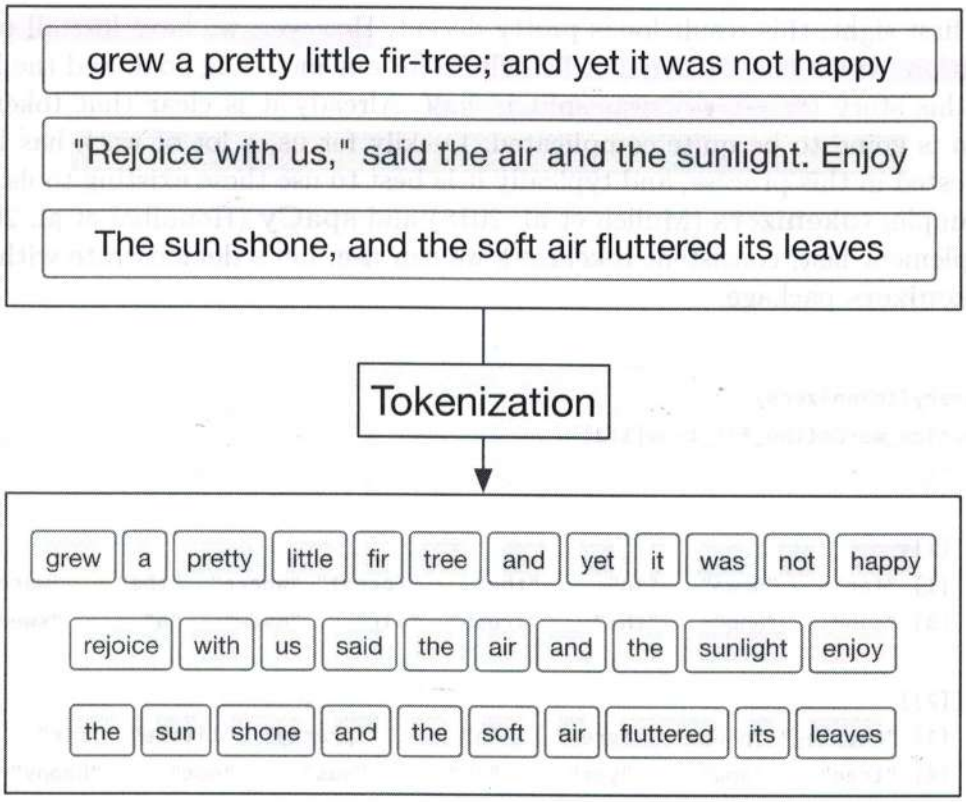


FIGURE 2.1: A black box representation of a tokenizer. The text of these three example text fragments has been converted to lowercase and punctuation has been removed before the text is split.

To understand the process of tokenization, let's start with a overly simple definition for a word: any selection of alphanumeric (letters and numbers) symbols. Let's use some regular expressions (or regex for short, see Appendix A) with `strsplit()` to split the first two lines of "The Fir-Tree" by any characters that are not alphanumeric.

```
strsplit(the_fir_tree[1:2], "[^a-zA-Z0-9]+")
```

```
#> [[1]]
#>  [1] "Far"    "down"   "in"     "the"    "forest" "where"  "the"    "warm"
#>  [9] "sun"    "and"    "the"    "fresh"  "air"    "made"   "a"      "sweet"
#>
#> [[2]]
#>  [1] "resting" "place"   "grew"    "a"       "pretty"  "little"  "fir"
#>  [8] "tree"    "and"     "yet"     "it"      "was"     "not"     "happy"
#> [15] "it"
```

At first sight, this result looks pretty decent. However, we have lost all punctuation, which may or may not be helpful for our modeling goal, and the hero of this story (`"fir-tree"`) was split in half. Already it is clear that tokenization is going to be quite complicated. Luckily for us, a lot of work has been invested in this process, and typically it is best to use these existing tools. For example, **tokenizers** (Mullen et al. 2018) and **spaCy** (Honnibal et al. 2020) implement fast, consistent tokenizers we can use. Let's demonstrate with the **tokenizers** package.

```
library(tokenizers)
tokenize_words(the_fir_tree[1:2])
```

```
#> [[1]]
#>  [1] "far"    "down"   "in"     "the"    "forest" "where"  "the"    "warm"
#>  [9] "sun"    "and"    "the"    "fresh"  "air"    "made"   "a"      "sweet"
#>
#> [[2]]
#>  [1] "resting" "place"   "grew"    "a"       "pretty"  "little"  "fir"
#>  [8] "tree"    "and"     "yet"     "it"      "was"     "not"     "happy"
#> [15] "it"
```

We see sensible single-word results here; the `tokenize_words()` function uses the **stringi** package (Gagolewski 2020) and C++ under the hood, making it very fast. Word-level tokenization is done by finding word boundaries according to the specification from the International Components for Unicode (ICU). How does this word boundary algorithm[1] work? It can be outlined as follows:

- Break at the start and end of text, unless the text is empty.
- Do not break within CRLF (new line characters).
- Otherwise, break before and after new lines (including CR and LF).
- Do not break within emoji zwj sequences.
- Keep horizontal whitespace together.
- Ignore Format and Extend characters, except after sot, CR, LF, and new lines.
- Do not break between most letters.

[1] https://www.unicode.org/reports/tr29/tr29-35.html#Default_Word_Boundaries

- Do not break letters across certain punctuation.
- Do not break within sequences of digits, or digits adjacent to letters ("3a," or "A3").
- Do not break within sequences, such as "3.2" or "3,456.789."
- Do not break between Katakana. (Japanese characters).
- Do not break from extenders.
- Do not break within emoji flag sequences.
- Otherwise, break everywhere (including around ideographs).

While we might not understand what each and every step in this algorithm is doing, we can appreciate that it is many times more sophisticated than our initial approach of splitting on non-alphanumeric characters. In most of this book, we will use the **tokenizers** package as a baseline tokenizer for reference. Your choice of tokenizer will influence your results, so don't be afraid to experiment with different tokenizers or, if necessary, to write your own to fit your problem.

! This is good to know when designing a tokenizer to work with Tolkien, who has lots of conjunction words, eg "pipe-weed" (just like "fir-tree").

2.2 Types of tokens

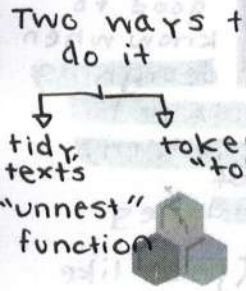
Thinking of a token as a word is a useful way to start understanding tokenization, even if it is hard to implement concretely in software. We can generalize the idea of a token beyond only a single word to other units of text. We can tokenize text at a variety of units including:

? Is there a package that does not split conjunction words? If not, how do I learn to make my own tokenizer

- characters,
- words,
- sentences,
- lines,
- paragraphs, and
- n-grams

In the following sections, we will explore how to tokenize text using the **tokenizers** package. These functions take a character vector as the input and return lists of character vectors as output. This same tokenization can also be done using the **tidytext** (Silge and Robinson 2016) package, for workflows using tidy data principles where the input and output are both in a dataframe.

```
sample_vector <- c("Far down in the forest",
                   "grew a pretty little fir-tree")
sample_tibble <- tibble(text = sample_vector)
```



The **tokenizers** package offers fast, consistent tokenization in R for tokens such as words, letters, n-grams, lines, paragraphs, and more.

The tokenization achieved by using `tokenize_words()` on `sample_vector`:

```
tokenize_words(sample_vector)
```

```
#> [[1]]
#> [1] "far"    "down"   "in"     "the"    "forest"
#>
#> [[2]]
#> [1] "grew"   "a"      "pretty" "little" "fir"    "tree"
```

will yield the same results as using `unnest_tokens()` on `sample_tibble`; the only difference is the data structure, and thus how we might use the result moving forward in our analysis.

```
sample_tibble %>%
  unnest_tokens(word, text, token = "words")
```

```
#> # A tibble: 11 x 1
#>    word
#>    <chr>
#>  1 far
#>  2 down
#>  3 in
#>  4 the
#>  5 forest
#>  6 grew
#>  7 a
#>  8 pretty
#>  9 little
#> 10 fir
#> 11 tree
```

(This is the tidy method)

The **tidytext** package provides functions to transform text to and from tidy formats, allowing us to work seamlessly with other **tidyverse** tools. ✓

Arguments used in `tokenize_words()` can be passed through `unnest_tokens()` using the "the dots"[2], ...

```
sample_tibble %>%
  unnest_tokens(word, text, token = "words", strip_punct = FALSE)
```

⇒ it "don't get rid of the punctuation"

```
#> # A tibble: 12 x 1
#>    word
#>    <chr>
#>  1 far
#>  2 down
#>  3 in
#>  4 the
#>  5 forest
#>  6 grew
```

✓

[2] https://adv-r.hadley.nz/functions.html#fun-dot-dot-dot

```
#>  7 a
#>  8 pretty
#>  9 little
#> 10 fir
#> 11 -
#> 12 tree
```

2.2.1 Character tokens

Perhaps the simplest tokenization is character tokenization, which splits texts into characters. Let's use `tokenize_characters()` with its default parameters; this function has arguments to convert to lowercase and to strip all non-alphanumeric characters. These defaults will reduce the number of different tokens that are returned. The `tokenize_*()` functions by default return a list of character vectors, one character vector for each string in the input.

```
tft_token_characters <- tokenize_characters(x = the_fir_tree,
                                            lowercase = TRUE,
                                            strip_non_alphanum = TRUE,
                                            simplify = FALSE)
```

What do we see if we take a look?

```
head(tft_token_characters) %>%
  glimpse()
```

```
#> List of 6
#>  $ : chr [1:57] "f" "a" "r" "d" ...
#>  $ : chr [1:57] "r" "e" "s" "t" ...
#>  $ : chr [1:61] "w" "i" "s" "h" ...
#>  $ : chr [1:56] "a" "r" "o" "u" ...
#>  $ : chr [1:64] "l" "i" "t" "t" ...
#>  $ : chr [1:64] "t" "h" "e" "m" ...
```

We don't have to stick with the defaults. We can keep the punctuation and spaces by setting `strip_non_alphanum = FALSE`, and now we see that spaces and punctuation are included in the results too.

```
tokenize_characters(x = the_fir_tree,
                    strip_non_alphanum = FALSE) %>%
  head() %>%
  glimpse()
```

```
#> List of 6
#>  $ : chr [1:73] "f" "a" "r" " " ...
#>  $ : chr [1:74] "r" "e" "s" "t" ...
#>  $ : chr [1:76] "w" "i" "s" "h" ...
#>  $ : chr [1:72] "a" "r" "o" "u" ...
#>  $ : chr [1:77] "l" "i" "t" "t" ...
#>  $ : chr [1:77] "t" "h" "e" "m" ...
```

The results have more elements because the spaces and punctuation have not been removed.

Depending on the format you have your text data in, it might contain ligatures. Ligatures are when multiple graphemes or letters are combined as a single character The graphemes "f" and "l" are combined into "fl," or "f" and "f" into "ff." When we apply normal tokenization rules the ligatures will not be split up.

```
tokenize_characters("flowers")
```

```
#> [[1]]
#> [1] "fl" "o" "w" "e" "r" "s"
```

We might want to have these ligatures separated back into separate characters, but first, we need to consider a couple of things. First, we need to consider if the presence of ligatures is a meaningful feature to the question we are trying to answer. Second, there are two main types of ligatures: stylistic and functional. Stylistic ligatures are when two characters are combined because the spacing between the characters has been deemed unpleasant. Functional ligatures like the German Eszett (also called the scharfes S, meaning sharp s) ß, is an official letter of the German alphabet. It is described as a long S and Z and historically has never gotten an uppercase character. This has led the typesetters to use SZ or SS as a replacement when writing a word in uppercase. Additionally, ß is omitted entirely in German writing in Switzerland and is replaced with ss. Other examples include the "W" in the Latin alphabet (two "v" or two "u" joined together), and æ, ø, and å in the Nordic languages. Some place names for historical reasons use the old spelling "aa" instead of å. In Section 6.7.1 we will discuss text normalization approaches to deal with ligatures.

2.2.2 Word tokens

Tokenizing at the word level is perhaps the most common and widely used tokenization. We started our discussion in this chapter with this kind of tokenization, and as we described before, this is the procedure of splitting text into words. To do this, let's use the `tokenize_words()` function.

```
tft_token_words <- tokenize_words(x = the_fir_tree,
                                  lowercase = TRUE,
                                  stopwords = NULL,
                                  strip_punct = TRUE,
                                  strip_numeric = FALSE)
```

The results show us the input text split into individual words.

```
head(tft_token_words) %>%
  glimpse()
```

```
#> List of 6
#>  $ : chr [1:16] "far" "down" "in" "the" ...
#>  $ : chr [1:15] "resting" "place" "grew" "a" ...
#>  $ : chr [1:15] "wished" "so" "much" "to" ...
#>  $ : chr [1:14] "around" "it" "the" "sun" ...
#>  $ : chr [1:12] "little" "peasant" "children" "passed" ...
#>  $ : chr [1:13] "them" "not" "sometimes" "the" ...
```

We have already seen `lowercase = TRUE`, and `strip_punct = TRUE` and `strip_numeric = FALSE` control whether we remove punctuation and numeric characters, respectively. We also have `stopwords = NULL`, which we will talk about in more depth in Chapter 3.

Let's create a tibble with two fairy tales, "The Fir-Tree" and "The Little Mermaid." Then we can use `unnest_tokens()` together with some **dplyr** verbs to find the most commonly used words in each.

```
hcandersen_en %>%
  filter(book %in% c("The fir tree", "The little mermaid")) %>%
  unnest_tokens(word, text) %>%
  count(book, word) %>%
  group_by(book) %>%
  arrange(desc(n)) %>%
  slice(1:5)
```

```
#> # A tibble: 10 x 3
#> # Groups:   book [2]
#>    book                word      n
#>    <chr>               <chr>  <int>
#>  1 The fir tree        the      278
#>  2 The fir tree        and      161
#>  3 The fir tree        tree      76
#>  4 The fir tree        it        66
#>  5 The fir tree        a         56
#>  6 The little mermaid  the      817
#>  7 The little mermaid  and      398
#>  8 The little mermaid  of       252
#>  9 The little mermaid  she      240
#> 10 The little mermaid  to       199
```

now many times that word appeared in said text

The five most common words in each fairy tale are fairly uninformative, with the exception being `"tree"` in the "The Fir-Tree."

These uninformative words are called **stop words** and will be explored in-depth in Chapter 3.

2.2.3 Tokenizing by n-grams

An n-gram (sometimes written "ngram") is a term in linguistics for a contiguous sequence of n items from a given sequence of text or speech. The item can be phonemes, syllables, letters, or words depending on the application, but when most people talk about n-grams, they mean a group of n words. In this book, we will use n-gram to denote word n-grams unless otherwise stated.

We use Latin prefixes so that a 1-gram is called a unigram, a 2-gram is called a bigram, a 3-gram called a trigram, and so on.

uni-, bi-, tri-, quad-, quin-, sex-, sept-...

Some example n-grams are:

- **unigram:** "Hello," "day," "my," "little"
- **bigram:** "fir tree," "fresh air," "to be," "Robin Hood"
- **trigram:** "You and I," "please let go," "no time like," "the little mermaid"

The benefit of using n-grams compared to words is that n-grams capture word order that would otherwise be lost. Similarly, when we use character n-grams, we can model the beginning and end of words, because a space will be located at the end of an n-gram for the end of a word and at the beginning of an n-gram of the beginning of a word.

To split text into word n-grams, we can use the function `tokenize_ngrams()`. It has a few more arguments, so let's go over them one by one.

```
tft_token_ngram <- tokenize_ngrams(x = the_fir_tree,
                                   lowercase = TRUE,
                                   n = 3,
                                   n_min = 3,
                                   stopwords = character(),
                                   ngram_delim = " ",
                                   simplify = FALSE)
```

We have seen the arguments `lowercase`, `stopwords`, and `simplify` before; they work the same as for the other tokenizers. We also have `n`, the argument to determine which degree of n-gram to return. Using `n = 1` returns unigrams, `n = 2` bigrams, `n = 3` gives trigrams, and so on. Related to `n` is the `n_min` argument, which specifies the minimum number of n-grams to include. By default both `n` and `n_min` are set to 3 making `tokenize_ngrams()` return only trigrams. By setting `n = 3` and `n_min = 1`, we will get all unigrams, bigrams, and trigrams of a text. Lastly, we have the `ngram_delim` argument, which specifies the separator between words in the n-grams; notice that this defaults to a space.

Let's look at the result of n-gram tokenization for the first line of "The Fir-Tree."

```
tft_token_ngram[[1]]
```

```
#>  [1] "far down in"      "down in the"      "in the forest"    "the forest where"
#>  [5] "forest where the" "where the warm"   "the warm sun"     "warm sun and"
#>  [9] "sun and the"      "and the fresh"    "the fresh air"    "fresh air made"
#> [13] "air made a"       "made a sweet"
```

Notice how the words in the trigrams overlap so that the word "down" appears in the middle of the first trigram and beginning of the second trigram. N-gram tokenization slides along the text to create overlapping sets of tokens.

It is important to choose the right value for `n` when using n-grams for the question we want to answer. Using unigrams is faster and more efficient, but we don't capture information about word order. Using a higher value for `n` keeps more information, but the vector space of tokens increases dramatically, corresponding to a reduction in token counts. A sensible starting point in most cases is three. However, if you don't have a large vocabulary in your data set, consider starting at two instead of three and experimenting from there. Figure 2.2 demonstrates how token frequency starts to decrease dramatically for trigrams and higher-order n-grams.

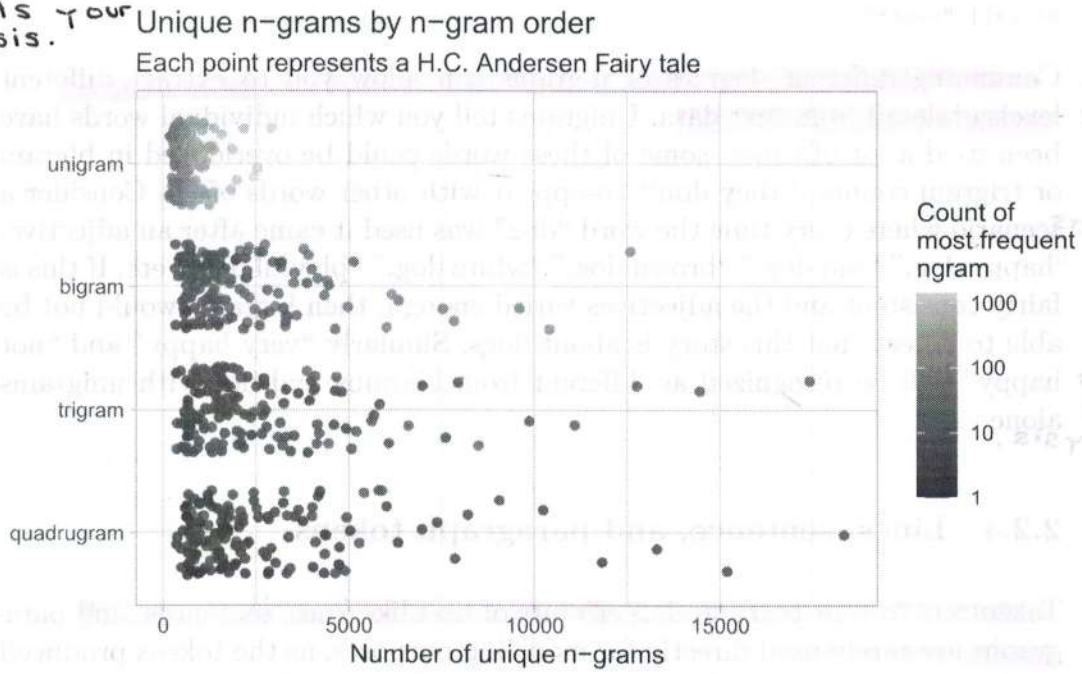


FIGURE 2.2: Using longer n-grams results in a higher number of unique tokens with fewer counts. Note that the color maps to counts on a logarithmic scale.

We are not limited to use only one degree of n-grams. We can, for example, combine unigrams and bigrams in an analysis or model. Getting multiple degrees of n-grams is a little different depending on what package you are using; using `tokenize_ngrams()` you can specify `n` and `n_min`.

```
tft_token_ngram <- tokenize_ngrams(x = the_fir_tree,
                                   n = 2L,
                                   n_min = 1L)
tft_token_ngram[[1]]
```

```
#>  [1] "far"          "far down"     "down"         "down in"      "in"
#>  [6] "in the"       "the"          "the forest"   "forest"       "forest where"
#> [11] "where"        "where the"    "the"          "the warm"     "warm"
#> [16] "warm sun"     "sun"          "sun and"      "and"          "and the"
#> [21] "the"          "the fresh"    "fresh"        "fresh air"    "air"
#> [26] "air made"     "made"         "made a"       "a"            "a sweet"
#> [31] "sweet"
```

Need to look into more linguistic & literary papers where they've used this method of combining different levels of n-grams into an analysis.

Combining different degrees of n-grams can allow you to extract different levels of detail from text data. Unigrams tell you which individual words have been used a lot of times; some of these words could be overlooked in bigram or trigram counts if they don't co-appear with other words often. Consider a scenario where every time the word "dog" was used it came after an adjective: "happy dog," "sad dog," "brown dog," "white dog," "playful dog," etc. If this is fairly consistent and the adjectives varied enough, then bigrams would not be able to detect that this story is about dogs. Similarly "very happy" and "not happy" will be recognized as different from bigrams and not with unigrams alone.

2.2.4 Lines, sentence, and paragraph tokens

Tokenizers to split text into larger units of text like lines, sentences, and paragraphs are rarely used directly for modeling purposes, as the tokens produced tend to be fairly unique. It is very uncommon for multiple sentences in a text to be identical! However, these tokenizers are useful for preprocessing and labeling.

For example, Jane Austen's novel *Northanger Abbey* (as available in the **janeaustenr** package) is already preprocessed with each line being at most 80 characters long. However, it might be useful to split the data into chapters and paragraphs instead.

ie tokenize text into paragraphs

Let's create a function that takes a dataframe containing a variable called `text` and turns it into a dataframe where the text is transformed into paragraphs. First, we can collapse the text into one long string using `collapse = "\n"` to denote line breaks, and then next we can use `tokenize_paragraphs()` to identify the paragraphs and put them back into a dataframe. We can add a paragraph count with `row_number()`.

```
add_paragraphs <- function(data) {
  pull(data, text) %>%
    paste(collapse = "\n") %>%
    tokenize_paragraphs() %>%
    unlist() %>%
    tibble(text = .) %>%
    mutate(paragraph = row_number())
}
```

Now we take the raw text data and add the chapter count by detecting when the characters `"CHAPTER"` appears at the beginning of a line. Then we `nest()` the text column, apply our `add_paragraphs()` function, and then `unnest()` again.

```
library(janeaustenr)

northangerabbey_paragraphed <- tibble(text = northangerabbey) %>%
  mutate(chapter = cumsum(str_detect(text, "^CHAPTER "))) %>%
  filter(chapter > 0,
         !str_detect(text, "^CHAPTER ")) %>%
  nest(data = text) %>%
  mutate(data = map(data, add_paragraphs)) %>%
  unnest(cols = c(data))

glimpse(northangerabbey_paragraphed)
```

```
#> Rows: 1,020
#> Columns: 3
#> $ chapter   <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, ~
#> $ text      <chr> "No one who had ever seen Catherine Morland in her infancy w~
#> $ paragraph <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1~
```

Now we have 1020 separate paragraphs we can analyze. Similarly, we could go a step further to split these chapters into sentences, lines, or words.

It can be useful to be able to reshape text data to get a different observational unit. As an example, if you wanted to build a sentiment classifier that would classify sentences as hostile or not, then you need to work with and train your model on sentences of text. Turning pages or paragraphs into sentences is a necessary step in your workflow.

Let us look at how we can turn `the_fir_tree` from a "one line per element" vector to a "one sentence per element." `the_fir_tree` comes as a vector so we start by using `paste()` to combine the lines back together. We use a space as

the separator, and then we pass it to the `tokenize_sentences()` function from the **tokenizers** package, which will perform sentence splitting.

```
the_fir_tree_sentences <- the_fir_tree %>%
  paste(collapse = " ") %>%
  tokenize_sentences()

head(the_fir_tree_sentences[[1]])
```

```
#> [1] "Far down in the forest, where the warm sun and the fresh air made a
sweet resting-place, grew a pretty little fir-tree; and yet it was not happy,
it wished so much to be tall like its companions— the pines and firs which grew
around it."
#> [2] "The sun shone, and the soft air fluttered its leaves, and the little
peasant children passed by, prattling merrily, but the fir-tree heeded them
not."
#> [3] "Sometimes the children would bring a large basket of raspberries or
strawberries, wreathed on a straw, and seat themselves near the fir-tree, and
say, \"Is it not a pretty little tree?\""
#> [4] "which made it feel more unhappy than before."
#> [5] "And yet all this while the tree grew a notch or joint taller every
year; for by the number of joints in the stem of a fir-tree we can discover its
age."
#> [6] "Still, as it grew, it complained."
```

If you have lines from different categories as we have in the `hcandersen_en` dataframe, which contains all the lines of the fairy tales in English, then we would like to be able to turn these lines into sentences while preserving the `book` column in the data set. To do this we use `nest()` and `map_chr()` to create a dataframe where each fairy tale is its own element and then we use the `unnest_sentences()` function from the tidytext package to split the text into sentences.

```
hcandersen_sentences <- hcandersen_en %>%
  nest(data = c(text)) %>%
  mutate(data = map_chr(data, ~ paste(.x$text, collapse = " "))) %>%
  unnest_sentences(sentences, data)
```

Now that we have turned the text into "one sentence per element," we can analyze on the sentence level.

2.3 Where does tokenization break down?

Tokenization will generally be one of the first steps when building a model or any kind of text analysis, so it is important to consider carefully what happens in this step of data preprocessing. As with most software, there is a trade-off between speed and customizability, as demonstrated in Section 2.6. The fastest tokenization methods give us less control over how it is done.

While the defaults work well in many cases, we encounter situations where we want to impose stricter rules to get better or different tokenized results. Consider the following sentence.

> Don't forget you owe the bank $1 million for the house.

This sentence has several interesting aspects that we need to decide whether to keep or to ignore when tokenizing. The first issue is the contraction in `"Don't"`, which presents us with several possible options. The fastest option is to keep this as one word, but it could also be split up into `"do"` and `"n't"`.

The next issue at hand is how to deal with `"$1"`; the dollar sign is an important part of this sentence as it denotes a kind of currency. We could either remove or keep this punctuation symbol, and if we keep the dollar sign, we can choose between keeping one or two tokens, `"$1"` or `"$"` and `"1"`. If we look at the default for `tokenize_words()`, we notice that it defaults to removing most punctuation including $.

```
tokenize_words("$1")
```

```
#> [[1]]
#> [1] "1"
```

We can keep the dollar sign if we don't strip punctuation.

```
tokenize_words("$1", strip_punct = FALSE)
```

```
#> [[1]]
#> [1] "$" "1"
```

When dealing with this sentence, we also need to decide whether to keep the final period as a token or not. If we remove it, we will not be able to locate the last word in a sentence using n-grams.

Information lost to tokenization (especially default tokenization) occurs more frequently in online and more casual text. Multiple spaces, extreme use of exclamation characters, and deliberate use of capitalization can be completely lost depending on our choice of tokenizer and tokenization parameters. At the same time, it is not always worth keeping that kind of information about how text is being used. If we are studying trends in disease epidemics using Twitter data, the style the tweets are written in is likely not nearly as important as what words are used. However, if we are trying to model social groupings, language style and how individuals use language toward each other becomes much more important.

Another thing to consider is the degree of compression each type of tokenization provides. The choice of tokenization results in a different pool of possible tokens and can influence performance. By choosing a method that gives fewer possible tokens you allow later computational tasks to be performed faster. However, that comes with the risk of collapsing together categories of a different meaning. It is also worth noting that the spread of the number of different tokens varies with your choice of tokenizer.

Figure 2.3 illustrates these points. Each of the fairy tales from **hcandersenr** has been tokenized in five different ways and the number of distinct tokens has been plotted along the x-axis (note that the x-axis is logarithmic). We see that the number of distinct tokens decreases if we convert words to lowercase or extract word stems (see Chapter 4 for more on stemming). Second, notice that the distributions of distinct tokens for character tokenizers are quite narrow; these texts use all or most of the letters in the English alphabet.

2.4 Building your own tokenizer

Sometimes the out-of-the-box tokenizers won't be able to do what you need them to do. In this case, we will have to wield **stringi**/**stringr** and regular expressions (see Appendix A).

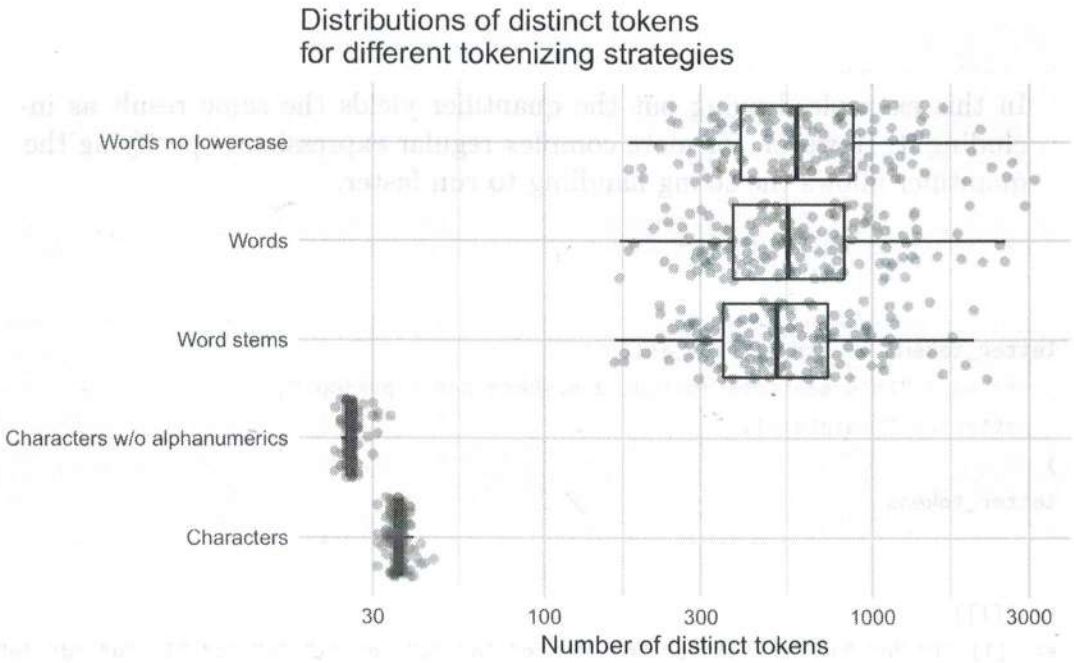


FIGURE 2.3: The number of distinct tokens can vary enormously for different tokenizers

There are two main approaches to tokenization.

1. *Split* the string up according to some rule.
2. *Extract* tokens based on some rule.

The number and complexity of our rules are determined by our desired outcome. We can reach complex outcomes by chaining together many smaller rules. In this section, we will implement a couple of specialty tokenizers to showcase these techniques.

2.4.1 Tokenize to characters, only keeping letters

Here we want to modify what `tokenize_characters()` does, such that we only keep letters. There are two main options. We can use `tokenize_characters()` and remove anything that is not a letter, or we can extract the letters one by one. Let's try the latter option. This is an **extract** task, and we will use `str_extract_all()` as each string has the possibility of including more than one token. Since we want to extract letters we can use the letters character class `[:alpha:]` to match letters and the quantifier `{1}` to only extract the first one.

In this example, leaving out the quantifier yields the same result as including it. However, for more complex regular expressions, specifying the quantifier allows the string handling to run faster.

```
letter_tokens <- str_extract_all(
  string = "This sentence include 2 numbers and 1 period.",
  pattern = "[:alpha:]{1}"
)
letter_tokens
```

```
#> [[1]]
#>  [1] "T" "h" "i" "s" "s" "e" "n" "t" "e" "n" "c" "e" "i" "n" "c" "l" "u" "d" "e"
#> [20] "n" "u" "m" "b" "e" "r" "s" "a" "n" "d" "p" "e" "r" "i" "o" "d"
```

We may be tempted to specify the character class as something like `[a-zA-Z]{1}`. This option would run faster, but we would lose non-English letter characters. This is a design choice we have to make depending on the goals of our specific problem.

```
danish_sentence <- "Så mødte han en gammel heks på landevejen"

str_extract_all(danish_sentence, "[:alpha:]")
```

```
#> [[1]]
#>  [1] "S" "å" "m" "ø" "d" "t" "e" "h" "a" "n" "e" "n" "g" "a" "m" "m" "e" "l" "h"
#> [20] "e" "k" "s" "p" "å" "l" "a" "n" "d" "e" "v" "e" "j" "e" "n"
```

```
str_extract_all(danish_sentence, "[a-zA-Z]")
```

```
#> [[1]]
#>  [1] "S" "m" "d" "t" "e" "h" "a" "n" "e" "n" "g" "a" "m" "m" "e" "l" "h" "e" "k"
#> [20] "s" "p" "l" "a" "n" "d" "e" "v" "e" "j" "e" "n"
```

Choosing between `[:alpha:]` and `[a-zA-Z]` may seem quite similar, but the resulting differences can have a big impact on your analysis.

2.4.2 Allow for hyphenated words

In our examples so far, we have noticed that the string "fir-tree" is typically split into two tokens. Let's explore two different approaches for how to handle this hyphenated word as one token. First, let's split on white space; this is a decent way to identify words in English and some other languages, and it does not split hyphenated words as the hyphen character isn't considered a white-space. Second, let's find a regex to match words with a hyphen and extract those.

Splitting by white space is not too difficult because we can use character classes, as shown in Table A.2. We will use the white space character class `[:space:]` to split our sentence.

```
str_split("This isn't a sentence with hyphenated-words.", "[:space:]")
```

```
#> [[1]]
#> [1] "This"          "isn't"          "a"
#> [4] "sentence"      "with"           "hyphenated-words."
```

This worked pretty well. This version doesn't drop punctuation, but we can achieve this by removing punctuation characters at the beginning and end of words.

```
str_split("This isn't a sentence with hyphenated-words.", "[:space:]") %>%
  map(~ str_remove_all(.x, "^[:punct:]+|[:punct:]+$"))
```

```
#> [[1]]
#> [1] "This"          "isn't"          "a"              "sentence"
#> [5] "with"          "hyphenated-words"
```

This regex used to remove the punctuation is a little complicated, so let's discuss it piece by piece.

- The regex `^[:punct:]+` will look at the beginning of the string (`^`) to match any punctuation characters (`[:punct:]`), where it will select one or more (`+`).

- The other regex `[:punct:]+$` will look for punctuation characters (`[:punct:]`) that appear one or more times (`+`) at the end of the string (`$`).

- These will alternate (`|`) so that we get matches from both sides of the words.

- The reason we use the quantifier `+` is that there are cases where a word is followed by multiple characters we don't want, such as `"okay..."` and `"Really?!!!"`.

We are using `map()` since `str_split()` returns a list, and we want `str_remove_all()` to be applied to each element in the list. (The example here only has one element.)

Now let's see if we can get the same result using extraction. We will start by constructing a regular expression that will capture hyphenated words; our definition here is a word with one hyphen located inside it. Since we want the hyphen to be inside the word, we will need to have a non-zero number of characters on either side of the hyphen.

This code won't catch 0 hyphenated words!

```
str_extract_all(
  string = "This isn't a sentence with hyphenated-words.",
  pattern = "[:alpha:]+-[:alpha:]+"
)
```

```
#> [[1]]
#> [1] "hyphenated-words"
```

but this one will

Wait, this only matched the hyphenated word! This happened because we are only matching words with hyphens. If we add the quantifier `?` then we can match 0 or 1 occurrences.

```
str_extract_all(
  string = "This isn't a sentence with hyphenated-words.",
  pattern = "[:alpha:]+-?[:alpha:]+"
)
```

```
#> [[1]]
#> [1] "This"             "isn"              "sentence"         "with"
#> [5] "hyphenated-words"
```

Now we are getting more words, but the ending of `"isn't"` is not there anymore and we lost the word `"a"`. We can get matches for the whole contraction by expanding the character class `[:alpha:]` to include the character `'`. We do that by using `[[:alpha:]']`.

```
str_extract_all(
  string = "This isn't a sentence with hyphenated-words.",
  pattern = "[[:alpha:]']+-?[[:alpha:]']+"
)
```

```
#> [[1]]
#> [1] "This"             "isn't"            "sentence"         "with"
#> [5] "hyphenated-words"
```

Next, we need to find out why `"a"` wasn't matched. If we look at the regular expression, we remember that we imposed the restriction that a non-zero number of characters needed to surround the hyphen to avoid matching words that start or end with a hyphen. This means that the smallest possible pattern matched is two characters long. We can fix this by using an alternation with `|`. We will keep our previous match on the left-hand side, and include `[:alpha:]{1}` on the right-hand side to match the single length words that won't be picked up by the left-hand side. Notice how we aren't using `[[:alpha:]']` since we are not interested in matching single `'` characters.

```
str_extract_all(
  string = "This isn't a sentence with hyphenated-words.",
  pattern = "[[:alpha:]']+-?[[:alpha:]']+|[:alpha:]{1}"
)
```

```
#> [[1]]
#> [1] "This"             "isn't"            "a"                "sentence"
#> [5] "with"             "hyphenated-words"
```

That is getting to be quite a complex regex, but we are now getting the same answer as before.

2.4.3 Wrapping it in a function

We have shown how we can use regular expressions to extract the tokens we want, perhaps to use in modeling. So far, the code has been rather unstructured. We would ideally wrap these tasks into functions that can be used the same way `tokenize_words()` is used.

Let's start with the example with hyphenated words. To make the function a little more flexible, let's add an option to transform all the output to lowercase.

```
tokenize_hyphenated_words <- function(x, lowercase = TRUE) {
  if (lowercase)
    x <- str_to_lower(x)

  str_split(x, "[:space:]") %>%
    map(~ str_remove_all(.x, "^[:punct:]+|[:punct:]+$"))
}

tokenize_hyphenated_words(the_fir_tree[1:3])
```

```
#> [[1]]
#>  [1] "far"    "down"   "in"     "the"    "forest" "where"  "the"    "warm"
#>  [9] "sun"    "and"    "the"    "fresh"  "air"    "made"   "a"      "sweet"
#>
#> [[2]]
#>  [1] "resting-place" "grew"          "a"             "pretty"
#>  [5] "little"        "fir-tree"      "and"           "yet"
#>  [9] "it"            "was"           "not"           "happy"
#> [13] "it"
#>
#> [[3]]
#>  [1] "wished"     "so"         "much"       "to"         "be"
#>  [6] "tall"       "like"       "its"        "companions" "the"
#> [11] "pines"      "and"        "firs"       "which"      "grew"
```

Notice how we transformed to lowercase first because the rest of the operations are case insensitive.

Next let's turn our character n-gram tokenizer into a function, with a variable `n` argument.

```
tokenize_character_ngram <- function(x, n) {
  ngram_loc <- str_locate_all(x, paste0("(?=(\\w{", n, "}))"))

  map2(ngram_loc, x, ~str_sub(.y, .x[, 1], .x[, 1] + n - 1))
}

tokenize_character_ngram(the_fir_tree[1:3], n = 3)
```

```
#> [[1]]
#>  [1] "Far" "dow" "own" "the" "for" "ore" "res" "est" "whe" "her" "ere" "the"
#> [13] "war" "arm" "sun" "and" "the" "fre" "res" "esh" "air" "mad" "ade" "swe"
#> [25] "wee" "eet"
#>
#> [[2]]
#>  [1] "res" "est" "sti" "tin" "ing" "pla" "lac" "ace" "gre" "rew" "pre" "ret"
#> [13] "ett" "tty" "lit" "itt" "ttl" "tle" "fir" "tre" "ree" "and" "yet" "was"
#> [25] "not" "hap" "app" "ppy"
#>
#> [[3]]
#>  [1] "wis" "ish" "she" "hed" "muc" "uch" "tal" "all" "lik" "ike" "its" "com"
#> [13] "omp" "mpa" "pan" "ani" "nio" "ion" "ons" "the" "pin" "ine" "nes" "and"
#> [25] "fir" "irs" "whi" "hic" "ich" "gre" "rew"
```

We can use `paste0()` in this function to construct an actual regex.

2.5 Tokenization for non-Latin alphabets

Our discussion of tokenization so far has focused on text where words are separated by white space and punctuation. For such text, even a quite basic tokenizer can give decent results. However, many written languages don't separate words in this way.

One of these languages is Chinese where each "word" can be represented by one or more consecutive characters. Splitting Chinese text into words is called "word segmentation" and is still an active area of research (Ma, Ganchev, and Weiss 2018; Huang et al. 2020).

We are not going to go into depth in this area, but we want to showcase that word segmentation is indeed possible with R as well. We use the **jiebaR** package (Wenfeng and Yanyi 2019). It is conceptually similar to the **tokenizers**

package, but we need to create a worker that is passed into `segment()` along with the string we want to segment.

```
library(jiebaR)
words <- c(" 下面是不分行输出的结果", " 下面是不输出的结果")

engine1 <- worker(bylines = TRUE)

segment(words, engine1)
```

```
#> [[1]]
#> [1] "下面" "是"   "不"   "分行" "输出" "的"   "结果"
#>
#> [[2]]
#> [1] "下面" "是"   "不"   "输出" "的"   "结果"
```

2.6 Tokenization benchmark

Not all tokenization packages are the same. Most open-source tokenizers in R are well-designed, but they are designed to serve different purposes. Some have a multitude of arguments to allow you to customize your tokenizer for greater flexibility, but this flexibility comes at a price; they tend to have relatively slower performance.

While we can't easily quantify flexibility, it is straightforward to benchmark some of the tokenizers available in R so you can pick the one that best suits your needs.

```
bench::mark(check = FALSE, iterations = 10,
  `corpus` = corpus::text_tokens(hcandersen_en$text),
  `tokenizers` = tokenizers::tokenize_words(hcandersen_en$text),
  `text2vec` = text2vec::word_tokenizer(hcandersen_en$text),
  `quanteda` = quanteda::tokenize_word(hcandersen_en$text),
  `base R` = strsplit(hcandersen_en$text, "\\s")
)
```

```
#> # A tibble: 5 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 corpus       89.5ms   91.6ms     10.4    12.21MB    2.61
#> 2 tokenizers  121.1ms  124.5ms      7.83    1.08MB    1.96
#> 3 text2vec     101ms    103ms      9.60   21.06MB    2.40
#> 4 quanteda    196.1ms  201.2ms      4.94    8.71MB    1.24
#> 5 base R      361.8ms  371.9ms      2.66   10.51MB    0.664
```

The corpus package (Perry 2020) offers excellent performance for tokenization, and other options are not much worse. One exception is using a base R function as a tokenizer; you will see significant performance gains by instead using a package built specifically for text tokenization.

2.7 Summary

To build a predictive model, text data needs to be split into meaningful units, called tokens. These tokens range from individual characters to words to n-grams and even more complex structures, and the particular procedure used to identify tokens from text can be important to your results. Fast and consistent tokenizers are available, but understanding how they behave and in what circumstances they work best will set you up for success. It's also possible to build custom tokenizers when necessary. Once text data is tokenized, a common next preprocessing step is to consider how to handle very common words that are not very informative— stop words. Chapter 3 examines this in detail.

2.7.1 In this chapter, you learned:

- that tokens are meaningful units of text, such as words or n-grams
- to implement different kinds of tokenization, the process of splitting text into tokens
- how different kinds of tokenization affect the distribution of tokens
- how to build your own tokenizer when the fast, consistent tokenizers that are available are not flexible enough

3

Stop words

Once we have split text into tokens, it often becomes clear that not all words carry the same amount of information, if any information at all, for a predictive modeling task. Common words that carry little (or perhaps no) meaningful information are called *stop words*. It is common advice and practice to remove stop words for various NLP tasks, but the task of stop word removal is more nuanced than many resources may lead you to believe. In this chapter, we will investigate what a stop word list is, the differences between them, and the effects of using them in your preprocessing workflow.

The concept of stop words has a long history with Hans Peter Luhn credited with coining the term in 1960 (Luhn 1960). Examples of these words in English are "a," "the," "of," and "didn't." These words are very common and typically don't add much to the meaning of a text but instead ensure the structure of a sentence is sound.

Categorizing words as either informative or non-informative is limiting, and we prefer to consider words as having a more fluid or continuous amount of information associated with them. This informativeness is context-specific as well. In fact, stop words themselves are often important in genre or authorship identification.

Historically, one of the main reasons for removing stop words was to decrease the computational time for text mining; it can be regarded as a dimensionality reduction of text data and was commonly-used in search engines to give better results (Huston and Croft 2010).

Stop words can have different roles in a corpus. We generally categorize stop words into three groups: global, subject, and document stop words.

Global stop words are words that are almost always low in meaning in a given language; these are words such as "of" and "and" in English that are needed to glue text together. These words are likely a safe bet for removal, but they are low in number. You can find some global stop words in pre-made stop word lists (Section 3.1).

DOI: 10.1201/9781003093459-3

Next up are subject-specific stop words. These words are uninformative for a given subject area. Subjects can be broad like finance and medicine or can be more specific like obituaries, health code violations, and job listings for librarians in Kansas. Words like "bath," "bedroom," and "entryway" are generally not considered stop words in English, but they may not provide much information for differentiating suburban house listings and could be subject stop words for certain analysis. You will likely need to manually construct such a stop word list (Section 3.2). These kinds of stop words may improve your performance if you have the domain expertise to create a good list.

Lastly, we have document-level stop words. These words do not provide any or much information for a given document. These are difficult to classify and won't be worth the trouble to identify. Even if you can find document stop words, it is not obvious how to incorporate this kind of information in a regression or classification task.

3.1 Using premade stop word lists

A quick option for using stop words is to get a list that has already been created. This is appealing because it is not difficult, but be aware that not all lists are created equal. Nothman, Qin, and Yurchak (2018) found some alarming results in a study of 52 stop word lists available in open-source software packages. Among some of the more grave issues were misspellings ("fify" instead of "fifty"), the inclusion of clearly informative words such as "computer" and "cry," and internal inconsistencies, such as including the word "has" but not the word "does." This is not to say that you should never use a stop word list that has been included in an open-source software project. However, you should always inspect and verify the list you are using, both to make sure it hasn't changed since you used it last, and also to check that it is appropriate for your use case.

There is a broad selection of stop word lists available today. For the purpose of this chapter, we will focus on three of the lists of English stop words provided by the **stopwords** package (Benoit, Muhr, and Watanabe 2021). The first is from the SMART (System for the Mechanical Analysis and Retrieval of Text) Information Retrieval System, an information retrieval system developed at Cornell University in the 1960s (Lewis et al. 2004). The second is the English Snowball stop word list (Porter 2001), and the last is the English list from the Stopwords ISO[1] collection. These stop word lists are all considered general purpose and not domain-specific.

[1] https://github.com/stopwords-iso/stopwords-iso

The **stopwords** package contains a comprehensive collection of stop word lists in one place for ease of use in analysis and other packages.

Before we start delving into the content inside the lists, let's take a look at how many words are included in each.

```
library(stopwords)
length(stopwords(source = "smart"))
length(stopwords(source = "snowball"))
length(stopwords(source = "stopwords-iso"))
```

```
#> [1] 571
#> [1] 175
#> [1] 1298
```

this'll help us see which package of stopwords to use...

The lengths of these lists are quite different, with the longest list being over seven times longer than the shortest! Let's examine the overlap of the words that appear in the three lists in an UpSet plot in Figure 3.1. An UpSet plot (Lex et al. 2014) visualizes intersections and aggregates of intersections of sets using a matrix layout, presenting the number of elements as well as summary statistics.

The UpSet plot in Figure 3.1 shows us that these three lists are almost true subsets of each other. The only exception is a set of 10 words that appear in Snowball and ISO but not in the SMART list. What are those words?

```
setdiff(stopwords(source = "snowball"),
        stopwords(source = "smart"))
```

```
#>  [1] "she's"   "he'd"    "she'd"   "he'll"   "she'll"  "shan't"  "mustn't"
#>  [8] "when's"  "why's"   "how's"
```

All these words are contractions. This is *not* because the SMART lexicon doesn't include contractions; if we look, there are almost 50 of them.

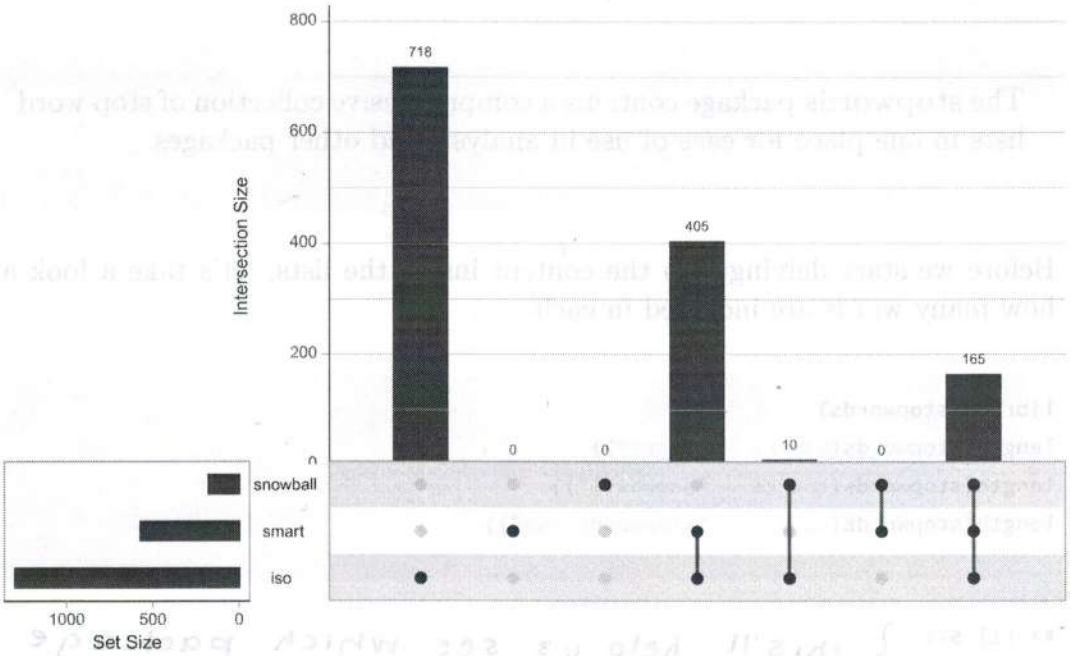


FIGURE 3.1: Set intersections for three common stop word lists visualized as an UpSet plot

```
str_subset(stopwords(source = "smart"), "'")
```

```
#>  [1] "a's"       "ain't"     "aren't"    "c'mon"     "c's"       "can't"
#>  [7] "couldn't"  "didn't"    "doesn't"   "don't"     "hadn't"    "hasn't"
#> [13] "haven't"   "he's"      "here's"    "i'd"       "i'll"      "i'm"
#> [19] "i've"      "isn't"     "it'd"      "it'll"     "it's"      "let's"
#> [25] "shouldn't" "t's"       "that's"    "there's"   "they'd"    "they'll"
#> [31] "they're"   "they've"   "wasn't"    "we'd"      "we'll"     "we're"
#> [37] "we've"     "weren't"   "what's"    "where's"   "who's"     "won't"
#> [43] "wouldn't"  "you'd"     "you'll"    "you're"    "you've"
```

We seem to have stumbled upon an inconsistency: why does SMART include `"he's"` but not `"she's"`? It is hard to say, but this could be worth rectifying before applying these stop word lists to an analysis or model preprocessing. This stop word list was likely generated by selecting the most frequent words across a large corpus of text that had more representation for text about men than women. This is once again a reminder that we should always look carefully at any pre-made word list or another artifact we use to make sure it works well with our needs[2].

[2]This advice applies to any kind of pre-made lexicon or word list, not just stop words. For instance, the same concerns apply to sentiment lexicons. The NRC sentiment lexicon of

It is perfectly acceptable to start with a premade word list and remove or append additional words according to your particular use case.

When you select a stop word list, it is important that you consider its size and breadth. Having a small and concise list of words can moderately reduce your token count while not having too great of an influence on your models, assuming that you picked appropriate words. As the size of your stop word list grows, each word added will have a diminishing positive effect with the increasing risk that a meaningful word has been placed on the list by mistake. In Section 6.4, we show the effects of different stop word lists on model training.

3.1.1 Stop word removal in R

Now that we have seen stop word lists, we can move forward with removing these words. The particular way we remove stop words depends on the shape of our data. If you have your text in a tidy format with one word per row, you can use `filter()` from **dplyr** with a negated `%in%` if you have the stop words as a vector, or you can use `anti_join()` from **dplyr** if the stop words are in a `tibble()`. Like in our previous chapter, let's examine the text of "The Fir-Tree" by Hans Christian Andersen, and use **tidytext** to tokenize the text into words.

```
library(hcandersenr)
library(tidyverse)
library(tidytext)

fir_tree <- hca_fairytales() %>%
  filter(book == "The fir tree",
         language == "English")

tidy_fir_tree <- fir_tree %>%
  unnest_tokens(word, text)
```

Let's use the Snowball stop word list as an example. Since the stop words return from this function as a vector, we will use `filter()`.

Mohammad and Turney (2013) associates the word "white" with trust and the word "black" with sadness, which could have unintended consequences when analyzing text about racial groups.

```
tidy_fir_tree %>%
  filter(!(word %in% stopwords(source = "snowball")))
```

```
#> # A tibble: 1,547 x 3
#>    book          language word
#>    <chr>         <chr>    <chr>
#>  1 The fir tree English  far
#>  2 The fir tree English  forest
#>  3 The fir tree English  warm
#>  4 The fir tree English  sun
#>  5 The fir tree English  fresh
#>  6 The fir tree English  air
#>  7 The fir tree English  made
#>  8 The fir tree English  sweet
#>  9 The fir tree English  resting
#> 10 The fir tree English  place
#> # ... with 1,537 more rows
```

If we use the `get_stopwords()` function from **tidytext** instead, then we can use the `anti_join()` function.

```
tidy_fir_tree %>%
  anti_join(get_stopwords(source = "snowball"))
```

```
#> # A tibble: 1,547 x 3
#>    book          language word
#>    <chr>         <chr>    <chr>
#>  1 The fir tree English  far
#>  2 The fir tree English  forest
#>  3 The fir tree English  warm
#>  4 The fir tree English  sun
#>  5 The fir tree English  fresh
#>  6 The fir tree English  air
#>  7 The fir tree English  made
#>  8 The fir tree English  sweet
#>  9 The fir tree English  resting
#> 10 The fir tree English  place
#> # ... with 1,537 more rows
```

The result of these two stop word removals is the same since we used the same stop word list in both cases.

3.2 Creating your own stop words list

Another way to get a stop word list is to create one yourself. Let's explore a few different ways to find appropriate words to use. We will use the tokenized data from "The Fir-Tree" as a first example. Let's take the words and rank them by their count or frequency.

Most frequent tokens in "The Fir-Tree"

1: the	25: said	49: trees	73: their	97: asked
2: and	26: what	50: we	74: which	98: can
3: tree	27: as	51: been	75: again	99: could
4: it	28: that	52: down	76: am	100: cried
5: a	29: he	53: oh	77: are	101: going
6: in	30: you	54: very	78: beautiful	102: grew
7: of	31: its	55: when	79: evening	103: if
8: to	32: out	56: where	80: him	104: large
9: i	33: be	57: who	81: like	105: looked
10: was	34: them	58: children	82: me	106: made
11: they	35: this	59: dumpty	83: more	107: many
12: fir	36: branches	60: humpty	84: about	108: seen
13: were	37: came	61: or	85: christmas	109: stairs
14: all	38: for	62: shall	86: do	110: think
15: with	39: now	63: there	87: fell	111: too
16: but	40: one	64: while	88: fresh	112: up
17: on	41: story	65: will	89: from	113: yes
18: then	42: would	66: after	90: here	114: air
19: had	43: forest	67: by	91: last	115: also
20: is	44: have	68: come	92: much	116: away
21: at	45: how	69: happy	93: no	117: birds
22: little	46: know	70: my	94: princess	118: corner
23: so	47: thought	71: old	95: tall	119: cut
24: not	48: mice	72: only	96: young	120: did

FIGURE 3.2: Words from "The Fir Tree" ordered by count or frequency

We recognize many of what we would consider stop words in the first column here, with three big exceptions. We see `"tree"` at 3, `"fir"` at 12, and `"little"` at 22. These words appear high on our list, but they do provide valuable information as they all reference the main character. What went wrong with this approach? Creating a stop word list using high-frequency words works best when it is created on a **corpus** of documents, not an individual document. This is because the words found in a single document will be document-specific and the overall pattern of words will not generalize that well.

In NLP, a corpus is a set of texts or documents. The set of Hans Christian Andersen's fairy tales can be considered a corpus, with each fairy tale a document within that corpus. The set of United States Supreme Court opinions can be considered a different corpus, with each written opinion being a document within *that* corpus. Both data sets are described in more detail in Appendix B.

The word `"tree"` does seem important as it is about the main character, but it could also be appearing so often that it stops providing any information. Let's try a different approach, extracting high-frequency words from the corpus of *all* English fairy tales by H.C. Andersen.

120 most frequent tokens in H.C. Andersen's English fairy tales

1: the	25: not	49: their	73: good	97: too
2: and	26: were	50: by	74: do	98: went
3: of	27: so	51: we	75: more	99: come
4: a	28: all	52: will	76: here	100: never
5: to	29: be	53: like	77: its	101: much
6: in	30: at	54: are	78: did	102: house
7: was	31: one	55: what	79: man	103: know
8: it	32: there	56: if	80: see	104: every
9: he	33: him	57: me	81: can	105: looked
10: that	34: from	58: up	82: through	106: many
11: i	35: have	59: very	83: beautiful	107: again
12: she	36: little	60: would	84: must	108: eyes
13: had	37: then	61: no	85: has	109: our
14: his	38: which	62: been	86: away	110: quite
15: they	39: them	63: about	87: thought	111: young
16: but	40: this	64: over	88: still	112: even
17: as	41: old	65: where	89: than	113: shall
18: her	42: out	66: an	90: well	114: tree
19: with	43: could	67: how	91: people	115: go
20: for	44: when	68: only	92: time	116: your
21: is	45: into	69: came	93: before	117: long
22: on	46: now	70: or	94: day	118: upon
23: said	47: who	71: down	95: other	119: two
24: you	48: my	72: great	96: stood	120: water

FIGURE 3.3: Words in all English fairy tales by Hans Christian Andersen ordered by count or frequency

This list is more appropriate for our concept of stop words, and now it is time for us to make some choices. How many do we want to include in our stop word list? Which words should we add and/or remove based on prior information? Selecting the number of words to remove is best done by a case-by-case basis as it can be difficult to determine a priori how many different "meaningless" words appear in a corpus. Our suggestion is to start with a low number like 20 and increase by 10 words until you get to words that are not appropriate as stop words for your analytical purpose.

It is worth keeping in mind that such a list is not perfect. Depending on how your text was generated or processed, strange tokens can surface as possible stop words due to encoding or optical character recognition errors. Further, these results are based on the corpus of documents we have available, which is potentially biased. In our example here, all the fairy tales were written by the same European white man from the early 1800s.

This bias can be minimized by removing words we would expect to be over-represented or to add words we expect to be under-represented.

Easy examples are to include the complements to the words in the list if they are not already present. Include "big" if "small" is present, "old" if "young" is present. This example list has words associated with women often listed lower in rank than words associated with men. With `"man"` being at rank 79 but `"woman"` at rank 179, choosing a threshold of 100 would lead to only one of these words being included. Depending on how important you think such nouns are going to be in your texts, consider either adding `"woman"` or deleting `"man"`.[3]

Figure 3.4 shows how the words associated with men have a higher rank than the words associated with women. By using a single threshold to create a stop word list, you would likely only include one form of such words.

Imagine now we would like to create a stop word list that spans multiple different genres, in such a way that the subject-specific stop words don't overlap. For this case, we would like words to be denoted as a stop word only if it is a stop word in all the genres. You could find the words individually in each genre and use the right intersections. However, that approach might take a substantial amount of time.

Below is a bad approach where we try to create a multi-language list of stop words. To accomplish this we calculate the *inverse document frequency*[4] (IDF) of each word. The IDF of a word is a quantity that is low for commonly-used words in a collection of documents and high for words not used often in a collection of documents. It is typically defined as

$$idf(\text{term}) = \ln\left(\frac{n_{\text{documents}}}{n_{\text{documents containing term}}}\right)$$

[3] On the other hand, the more biased stop word list may be helpful when modeling a corpus with gender imbalance, depending on your goal; words like "she" and "her" can identify where women are mentioned.

[4] `https://www.tidytextmining.com/tfidf.html`

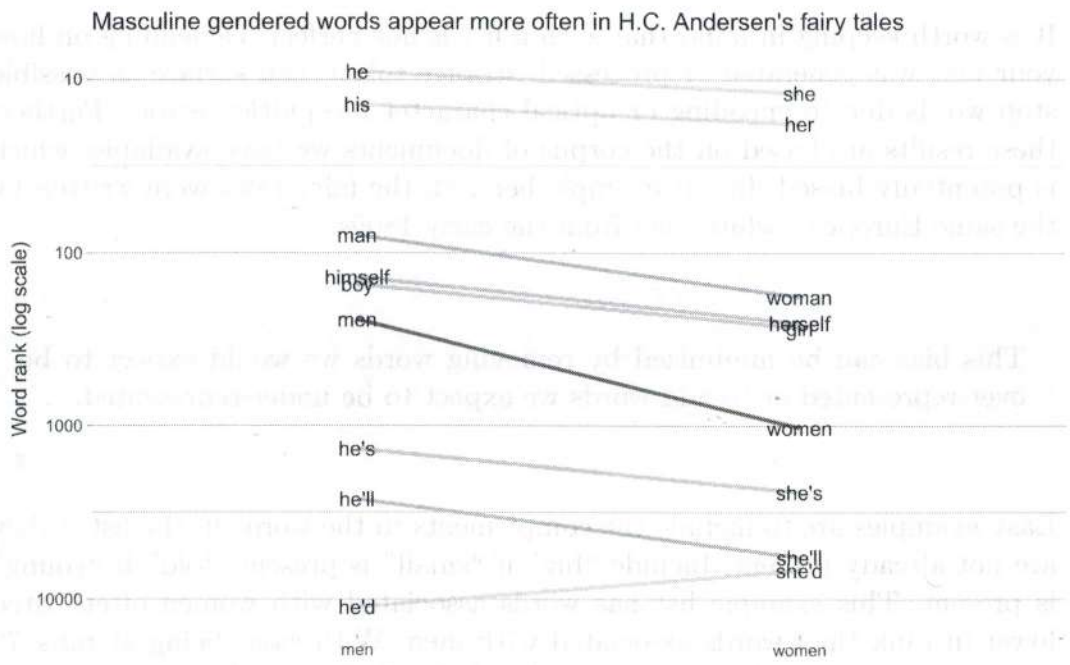


FIGURE 3.4: Tokens ranked according to total occurrences, with rank 1 having the most occurrences

If the word "dog" appears in 4 out of 100 documents then it would have an `idf("dog") = log(100/4) = 3.22`, and if the word "cat" appears in 99 out of 100 documents then it would have an `idf("cat") = log(100/99) = 0.01`. Notice how the idf values goes to zero (as a matter of fact when a term appears in all the documents then the idf of that word is 0 `log(100/100) = log(1) = 0`), the more documents it is contained in. What happens if we create a stop word list based on words with the lowest IDF? The following function takes a tokenized dataframe and returns a dataframe with a column for each word and a column for the IDF.

```
library(rlang)
calc_idf <- function(df, word, document) {
  words <- df %>% pull({{word}}) %>% unique()
  n_docs <- length(unique(pull(df, {{document}})))
  n_words <- df %>%
    nest(data = c({{word}})) %>%
    pull(data) %>%
    map_dfc(~ words %in% unique(pull(.x, {{word}}))) %>%
    rowSums()

  tibble(word = words,
```

```
        idf = log(n_docs / n_words))
}
```

Here is the result when we try to create a cross-language list of stop words, by taking each fairy tale as a document. It is not very good!

The overlap between words that appear in each language is very small, but these words are what we mostly see in this list.

120 tokens in H.C. Andersen's fairy tales with lowest IDF, multi-language

1: a	25: he	49: ser	73: and	97: pero
2: de	26: alle	50: et	74: o	98: them
3: man	27: ja	51: lo	75: alt	99: had
4: en	28: have	52: die	76: war	100: vi
5: da	29: to	53: just	77: ni	101: das
6: se	30: mit	54: bien	78: su	102: his
7: es	31: all	55: vor	79: time	103: les
8: an	32: oh	56: las	80: von	104: sagte
9: in	33: will	57: del	81: hand	105: ist
10: her	34: am	58: still	82: the	106: ein
11: me	35: la	59: land	83: that	107: und
12: so	36: sang	60: under	84: it	108: zu
13: no	37: le	61: has	85: of	109: para
14: i	38: des	62: los	86: there	110: sol
15: for	39: y	63: by	87: sit	111: auf
16: den	40: un	64: as	88: with	112: sie
17: at	41: que	65: not	89: por	113: nicht
18: der	42: on	66: end	90: el	114: aber
19: was	43: men	67: fast	91: con	115: sich
20: du	44: stand	68: hat	92: una	116: then
21: er	45: al	69: see	93: be	117: were
22: dem	46: si	70: but	94: they	118: said
23: over	47: son	71: from	95: one	119: into
24: sin	48: han	72: is	96: como	120: más

FIGURE 3.5: Words from all of H.C. Andersen's fairy tales in Danish, English, French, German, and Spanish, counted and ordered by IDF

This didn't work very well because there is very little overlap between common words. Instead, let us limit the calculation to only one language and calculate the IDF of each word we can find compared to words that appear in a lot of documents.

120 tokens in H.C. Andersen's fairy tales with lowest IDF, English only

1: a	25: them	49: if	73: good	97: own
2: the	26: be	50: little	74: must	98: come
3: and	27: from	51: over	75: my	99: its
4: to	28: had	52: are	76: than	100: whole
5: in	29: then	53: very	77: away	101: just
6: that	30: were	54: you	78: more	102: many
7: it	31: said	55: him	79: has	103: never
8: but	32: into	56: we	80: thought	104: made
9: of	33: by	57: great	81: did	105: stood
10: was	34: have	58: how	82: other	106: yet
11: as	35: which	59: their	83: still	107: looked
12: there	36: this	60: came	84: do	108: again
13: on	37: up	61: been	85: even	109: say
14: at	38: out	62: down	86: before	110: may
15: is	39: what	63: would	87: me	111: yes
16: for	40: who	64: where	88: know	112: went
17: with	41: no	65: or	89: much	113: every
18: all	42: an	66: she	90: see	114: each
19: not	43: now	67: can	91: here	115: such
20: they	44: i	68: could	92: well	116: world
21: one	45: only	69: about	93: through	117: some
22: he	46: old	70: her	94: day	118: long
23: his	47: like	71: will	95: too	119: eyes
24: so	48: when	72: time	96: people	120: go

FIGURE 3.6: Words from all of H.C. Andersen's fairy tales in English, counted and ordered by IDF

This time we get better results. The list starts with "a," "the," "and," and "to" and continues with many more reasonable choices of stop words. We need to look at these results manually to turn this into a list. We need to go as far down in rank as we are comfortable with. You as a data practitioner are in full control of how you want to create the list. If you don't want to include "little" you are still able to add "are" to your list even though it is lower on the list.

3.3 All stop word lists are context-specific

Context is important in text modeling, so it is important to ensure that the stop word lexicon you use reflects the word space that you are planning on using it in. One common concern to consider is how pronouns bring information to your text. Pronouns are included in many different stop word lists (although inconsistently), but they will often *not* be noise in text data. Similarly, Bender et al. (2021) discuss how a list of about 400 "Dirty, Naughty, Obscene or Otherwise Bad Words" were used to filter and remove text before training a trillion parameter large language model, to protect it from learning offensive

language, but the authors point out that in some community contexts, such words are reclaimed or used to describe marginalized identities.

On the other hand, sometimes you will have to add in words yourself, depending on the domain. If you are working with texts for dessert recipes, certain ingredients (sugar, eggs, water) and actions (whisking, baking, stirring) may be frequent enough to pass your stop word threshold, but you may want to keep them as they may be informative. Throwing away "eggs" as a common word would make it harder or downright impossible to determine if certain recipes are vegan or not while whisking and stirring may be fine to remove as distinguishing between recipes that do and don't require a whisk might not be that big of a deal.

3.4 What happens when you remove stop words

We have discussed different ways of finding and removing stop words; now let's see what happens once you do remove them. First, let's explore the impact of the number of words that are included in the list. Figure 3.7 shows what percentage of words are removed as a function of the number of words in a text. The different colors represent the three different stop word lists we have considered in this chapter.

We notice, as we would predict, that larger stop word lists remove more words than shorter stop word lists. In this example with fairy tales, over half of the words have been removed, with the largest list removing over 80% of the words. We observe that shorter texts have a lower percentage of stop words. Since we are looking at fairy tales, this could be explained by the fact that a story has to be told regardless of the length of the fairy tale, so shorter texts are going to be denser with more informative words.

Another problem you may face is dealing with misspellings.

> Most premade stop word lists assume that all the words are spelled correctly.

Handling misspellings when using premade lists can be done by manually adding common misspellings. You could imagine creating all words that are a certain string distance away from the stop words, but we do not recommend this as you would quickly include informative words this way.

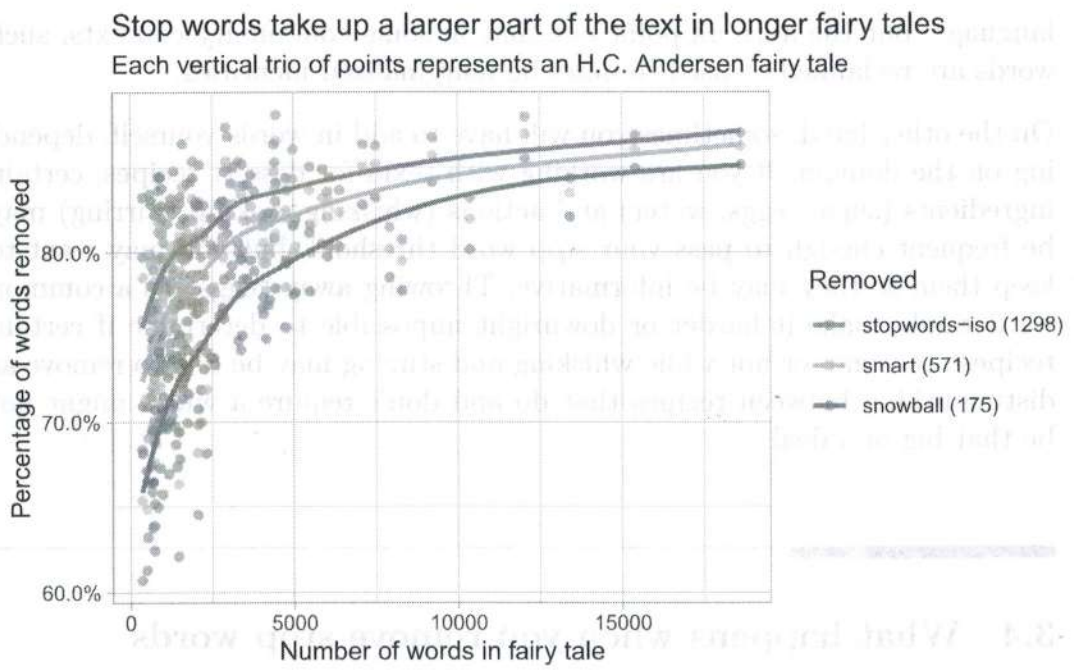


FIGURE 3.7: Proportion of words removed for different stop word lists and different document lengths

One of the downsides of creating your own stop word lists using frequencies is that you are limited to using words that you have already observed. It could happen that "she'd" is included in your training corpus but the word "he'd" did not reach the threshold. This is a case where you need to look at your words and adjust accordingly. Here the large premade stop word lists can serve as inspiration for missing words.

In Section 6.4, we investigate the influence of removing stop words in the context of modeling. Given the right list of words, we see no harm to the model performance, and sometimes find improvement due to noise reduction (Feldman, and Sanger 2007).

3.5 Stop words in languages other than English

So far in this chapter, we have focused on English stop words, but English is not representative of every language. The notion of "short" and "long" lists we have used so far are specific to English as a language. You should expect different languages to have a different number of "uninformative" words, and for this number to depend on the morphological richness of a language; lists that

contain all possible morphological variants of each stop word could become quite large.

Different languages have different numbers of words in each class of words. An example is how the grammatical case influences the articles used in German. The following tables show the use of definite and indefinite articles in German[5]. Notice how German nouns have three genders (masculine, feminine, and neuter), which are not uncommon in languages around the world. Articles are almost always considered to be stop words in English as they carry very little information. However, German articles give some indication of the case, which can be used when selecting a list of stop words in German.

German Definite Articles (the)

	Masculine	Feminine	Neuter	Plural
Nominative	der	die	das	die
Accusative	den	die	das	die
Dative	dem	der	dem	den
Genitive	des	der	des	der

German Indefinite Articles (a/an)

	Masculine	Feminine	Neuter
Nominative	ein	eine	ein
Accusative	einen	eine	ein
Dative	einem	einer	einem
Genitive	eines	einer	eines

Building lists of stop words in Chinese has been done both manually and automatically (Zou, Wang, Deng, Han, and Wang 2006) but so far none has been accepted as a standard (Zou, Wang, Deng, and Han 2006). A full discussion of stop word identification in Chinese text would be out of scope for this book, so we will just highlight some of the challenges that differentiate it from English.

Chinese text is much more complex than portrayed here. With different systems and billions of users, there is much we won't be able to touch on here.

[5] https://deutsch.lingolia.com/en/grammar/nouns-and-articles/articles-noun-markers

The main difference from English is the use of logograms instead of letters to convey information. However, Chinese characters should not be confused with Chinese words. The majority of words in modern Chinese are composed of multiple characters. This means that inferring the presence of words is more complicated, and the notion of stop words will affect how this segmentation of characters is done.

3.6 Summary

In many standard NLP workflows, the removal of stop words is presented as a default or the correct choice without comment. Although removing stop words can improve the accuracy of your machine learning using text data, choices around such a step are complex. The content of existing stop word lists varies tremendously, and the available strategies for building your own can have subtle to not-so-subtle effects on your model results.

3.6.1 In this chapter, you learned:

- what a stop word is and how to remove stop words from text data
- how different stop word lists can vary
- that the impact of stop word removal is different for different kinds of texts
- about the bias built in to stop word lists and strategies for building such lists

It's great that these first 3 chapters really outlined the limits to tokenization methods & stopword methods, especially in relation to non-EN languages, and shows us how to customize. It's useful for my research project in Tolkien's LOTR and TH.