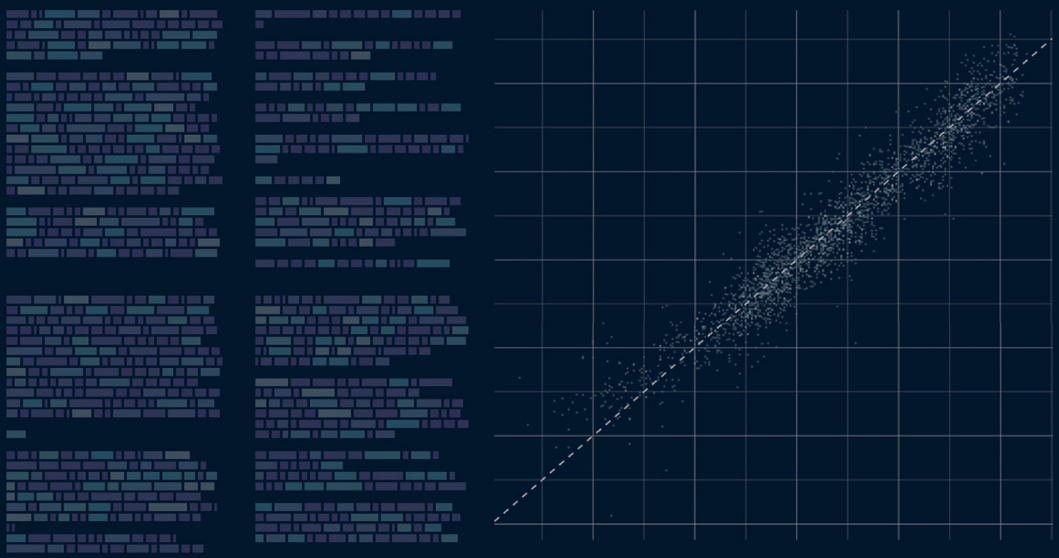
DATA SCIENCE SERIES

SUPERVISED MACHINE LEARNING FOR TEXT ANALYSIS IN R



EMIL HVITFELDT
JULIA SILGE

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Supervised Machine Learning for Text Analysis in R

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Supervised Machine Learning for Text Analysis in R

Emil Hvitfeldt
Julia Silge

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

First edition published 2022
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-0-367-55418-7 (hbk)
ISBN: 978-0-367-55419-4 (pbk)
ISBN: 978-1-003-09345-9 (ebk)

DOI: 10.1201/9781003093459

Typeset in LMR10 font
by KnowledgeWorks Global Ltd.

In loving memory of my mother-in-law Lisa, who was the first soul to hear about and fully encourage the idea that eventually became this book —E.H.

For Grace, Violet, and Lewis, who (thanks to the pandemic and remote school) had a front row seat to most of my work on this book —J.S.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface

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.

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.

- **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 (@tmstauss).

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)

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Part I

Natural Language Features

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

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.

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.

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

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

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.

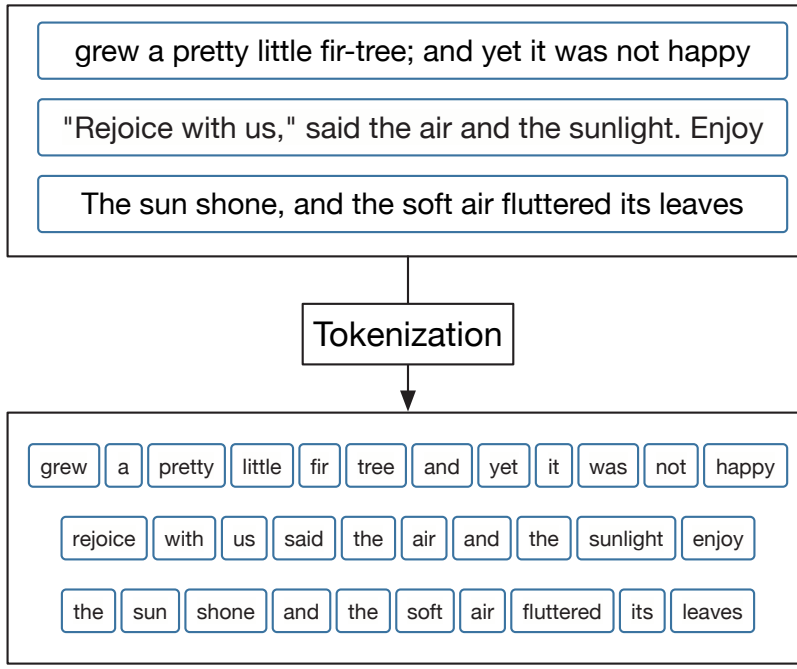


FIGURE 2.1: A black box representation of a tokenizer. The text of these three example text fragments has been converted to lowercase and punctuation has been removed before the text is split.

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.
- 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.

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:

- 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
```

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)
```

```
#> # 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
```

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.

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 = 3L,
                                   n_min = 3L,
                                   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"
```