Quantitative Methods in the Humanities and Social Sciences

Guillaume Desagulier

Corpus Linguistics and Statistics with R

Introduction to Quantitative Methods in Linguistics

Springer

Quantitative Methods in the Humanities and Social Sciences

Quantitative Methods in the Humanities and Social Sciences is a book series designed to foster research-based conversation with all parts of the university campus from buildings of ivy-covered stone to technologically savvy walls of glass. Scholarship from international researchers and the esteemed editorial board represents the far-reaching applications of computational analysis, statistical models, computer-based programs, and other quantitative methods. Methods are integrated in a dialogue that is sensitive to the broader context of humanistic study and social science research. Scholars, including among others historians, archaeologists, classicists and linguists, promote this interdisciplinary approach. These texts teach new methodological approaches for contemporary research. Each volume exposes readers to a particular research method. Researchers and students then benefit from exposure to subtleties of the larger project or corpus of work in which the quantitative methods come to fruition.

More information about this series at http://www.springer.com/series/11748

Guillaume Desagulier

Corpus Linguistics and Statistics with R

Introduction to Quantitative Methods in Linguistics

Springer

Guillaume Desagulier
Université Paris 8
Saint Denis, France

Additional material to this book can be downloaded from http://extras.springer.com.

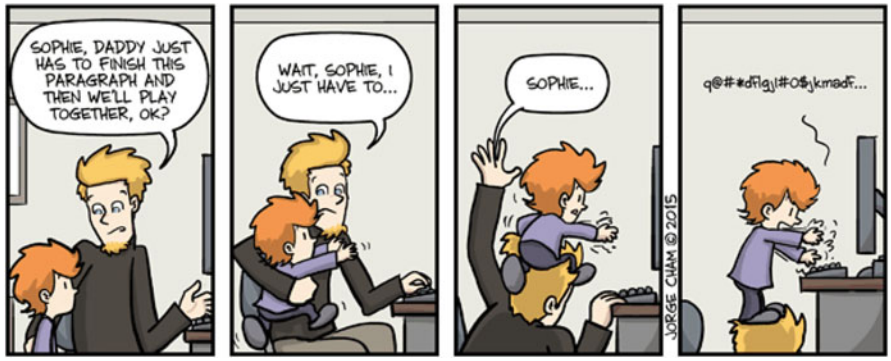
ISSN 2199-0956 ISSN 2199-0964 (electronic)
Quantitative Methods in the Humanities and Social Sciences
ISBN 978-3-319-64570-4 ISBN 978-3-319-64572-8 (eBook)
DOI 10.1007/978-3-319-64572-8

Library of Congress Control Number: 2017950518

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Fatima, Idris, and Hanaé (who knows how to finish a paragraph).



"Piled Higher and Deeper" by Jorge Cham, www.phdcomics.com

Preface

Who Should Read This Book

In the summer of 2008, I gave a talk at an international conference in Brighton. The talk was about constructions involving multiple hedging in American English (e.g., *I'm gonna have to ask you to* + VP). I remember this talk because even though I had every reason to be happy (the audience showed sustained interest and a major linguist in my field gave me positive feedback), I remember feeling a pang of dissatisfaction. Because my research was mostly theoretical at the time, I had concluded my presentation with the phrase "pending empirical validation" one too many times. Of course, I had used examples gleaned from the renowned *Corpus of Contemporary American English*, but my sampling was not exhaustive and certainly biased. Even though I felt I had answered my research questions, I had provided no quantitative summary. I went home convinced that it was time to buttress my research with corpus data. I craved for a better understanding of corpora, their constitution, their assets, and their limits. I also wished to extract the data I needed the way I wanted, beyond what traditional, prepackaged corpus tools have to offer.

I soon realized that the kind of corpus linguistics that I was looking for was technically demanding, especially for the armchair linguist that I was. In the summer of 2010, my lab offered that I attend a one-week boot camp in Texas whose instructor, Stefan Th. Gries (University of Santa Barbara), had just published *Quantitative Corpus Linguistics with R*. This boot camp was a career-changing opportunity. I went on to teach myself more elaborate corpus-linguistics techniques as well as the kinds of statistics that linguists generally have other colleagues do for them. This led me to collaborate with great people outside my field, such as mathematicians, computer engineers, and experimental linguists. While doing so, I never left aside my research in theoretical linguistics. I can say that acquiring empirical skills has made me a better theoretical linguist.

If the above lines echo your own experience, this book is perfect for you. While written for a readership with little or no background in corpus linguistics, computer programming, or statistics, *Corpus Linguistics and Statistics with R* will also appeal to readers with more experience in these fields. Indeed, while presenting in detail the text-mining apparatus used in traditional corpus linguistics (frequency lists, concordance tables, collocations, etc.), the text also introduces the reader to some appealing techniques that I wish I had become acquainted with much earlier in my career (motion charts, word clouds, network graphs, etc.).

Goals

This is a book on empirical linguistics written from a theoretical linguist's perspective. It provides both a theoretical discussion of what quantitative corpus linguistics entails and detailed, hands-on, step-by-step instructions to implement the techniques in the field.

A Note to Instructors

I have written this book so that instructors feel free to teach the chapters individually in the order that they want. For a one-semester course, the emphasis can be on either:

- Methods in corpus linguistics (Part I)
- Statistics for corpus linguistics (Part II)

In either case, I would recommend to always include Chaps. 1 and 2 to make sure that the students are familiar with the core concepts of corpus-based research and R programming.

Note that it is also possible to include all the chapters in a one-semester course, as I do on a regular basis. Be aware, however, that this leaves the students less time to apply the techniques to their own research projects. Experience tells me that the students need sufficient time to work through all of the earlier chapters, as well as to accommodate to the statistical analysis material.

Supplementary Materials

Through the chapters, readers will make extensive use of datasets and code. These materials are available from the book's Springer Extra's website:

`http://extras.springer.com/2017/978-3-319-64570-4`

I recommend that you check the repository on a regular basis for updates.

Acknowledgments

Writing this book would not have been possible without the help and support of my family, friends, and colleagues. Thanks go to Fatima, Idris, and Hanaé for their patience. In particular, I would like to thank those who agreed to proofread and test-drive early drafts in real-life conditions. Special thanks go to Antoine Chambaz for his insightful comments on Chap. 8.

Paris, France
April 2017

Guillaume Desagulier

Contents

1 Introduction 1
1.1 From Introspective to Corpus-Informed Judgments 1
1.2 Looking for Corpus Linguistics 3
1.2.1 What Counts as a Corpus 3
1.2.2 What Linguists Do with the Corpus 6
1.2.3 How Central the Corpus Is to a Linguist's Work 8
References 10

Part I Methods in Corpus Linguistics

2 R Fundamentals 15
2.1 Introduction 15
2.2 Downloads and Installs 15
2.2.1 Downloading and Installing R 16
2.2.2 Downloading and Installing RStudio 16
2.2.3 Downloading the Book Materials 17
2.3 Setting the Working Directory 17
2.4 R Scripts 17
2.5 Packages 18
2.5.1 Downloading Packages 18
2.5.2 Loading Packages 19
2.6 Simple Commands 19
2.7 Variables and Assignment 20
2.8 Functions and Arguments 21
2.8.1 Ready-Made Functions 21
2.8.2 User-Defined Functions 22
2.9 R Objects 24
2.9.1 Vectors 24
2.9.2 Lists 33
2.9.3 Matrices 34
2.9.4 Data Frames (and Factors) 36
2.10 `for` Loops 41
2.11 `if` and `if...else` Statements 43

2.11.1 `if` Statements . . . 43
2.11.2 `if...else` Statements . . . 44
2.12 Cleanup . . . 45
2.13 Common Mistakes and How to Avoid Them . . . 46
2.14 Further Reading . . . 47
Exercises . . . 47
References . . . 49

3 Digital Corpora . . . 51
3.1 A Short Typology . . . 51
3.2 Corpus Compilation: Kennedy's Five Steps . . . 52
3.3 Unannotated Corpora . . . 54
3.3.1 Collecting Textual Data . . . 54
3.3.2 Character Encoding Issues . . . 55
3.3.3 Creating an Unannotated Corpus . . . 57
3.4 Annotated Corpora . . . 58
3.4.1 Markup . . . 58
3.4.2 POS-Tagging . . . 58
3.4.3 POS-Tagging in R . . . 59
3.4.4 Semantic Tagging . . . 63
3.5 Obtaining Corpora . . . 65
Exercise . . . 65
References . . . 66

4 Processing and Manipulating Character Strings . . . 69
4.1 Introduction . . . 69
4.2 Character Strings . . . 69
4.2.1 Definition . . . 70
4.2.2 Loading Several Text Files . . . 70
4.3 First Forays into Character String Processing . . . 71
4.3.1 Splitting . . . 71
4.3.2 Matching . . . 72
4.3.3 Replacing and Deleting . . . 72
4.3.4 Limitations . . . 73
4.4 Regular Expressions . . . 73
4.4.1 Overview . . . 73
4.4.2 Literals vs. Metacharacters . . . 74
4.4.3 Line Anchors . . . 74
4.4.4 Quantifiers . . . 75
4.4.5 Alternations and Groupings . . . 76
4.4.6 Character Classes . . . 77
4.4.7 Lazy vs. Greedy Matching . . . 79
4.4.8 Backreference . . . 80
4.4.9 Exact Matching with `strapply()` . . . 81
4.4.10 Lookaround . . . 82
Exercises . . . 85

5 Applied Character String Processing . . . 87
5.1 Introduction . . . 87
5.2 Concordances . . . 87
5.2.1 A Concordance Based on an Unannotated Corpus . . . 87
5.2.2 A Concordance Based on an Annotated Corpus . . . 95
5.3 Making a Data Frame from an Annotated Corpus . . . 104
5.3.1 Planning the Data Frame . . . 104
5.3.2 Compiling the Data Frame . . . 104
5.3.3 The Full Script . . . 106
5.4 Frequency Lists . . . 108
5.4.1 A Frequency List of a Raw Text File . . . 108
5.4.2 A Frequency List of an Annotated File . . . 110
Exercises . . . 113
References . . . 114

6 Summary Graphics for Frequency Data . . . 115
6.1 Introduction . . . 115
6.2 Plots, Barplots, and Histograms . . . 115
6.3 Word Clouds . . . 118
6.4 Dispersion Plots . . . 122
6.5 Strip Charts . . . 125
6.6 Reshaping Tabulated Data . . . 127
6.7 Motion Charts . . . 132
Exercises . . . 133
References . . . 135

Part II Statistics for Corpus Linguistics

7 Descriptive Statistics . . . 139
7.1 Variables . . . 139
7.2 Central Tendency . . . 140
7.2.1 The Mean . . . 140
7.2.2 The Median . . . 142
7.2.3 The Mode . . . 143
7.3 Dispersion . . . 145
7.3.1 Quantiles . . . 145
7.3.2 Boxplots . . . 146
7.3.3 Variance and Standard Deviation . . . 147
Exercises . . . 148

8 Notions of Statistical Testing . . . 151
8.1 Introduction . . . 151
8.2 Probabilities . . . 151
8.2.1 Definition . . . 151
8.2.2 Simple Probabilities . . . 152
8.2.3 Joint and Marginal Probabilities . . . 153
8.2.4 Union vs. Intersection . . . 155

8.2.5 Conditional Probabilities . . . 155
8.2.6 Independence . . . 156
8.3 Populations, Samples, and Individuals . . . 157
8.4 Random Variables . . . 158
8.5 Response/Dependent vs. Explanatory/Descriptive/Independent Variables . . . 159
8.6 Hypotheses . . . 160
8.7 Hypothesis Testing . . . 162
8.8 Probability Distributions . . . 163
8.8.1 Discrete Distributions . . . 165
8.8.2 Continuous Distributions . . . 169
8.9 The χ^2 Test . . . 178
8.9.1 A Case Study: The Quotative System in British and Canadian Youth . . . 178
8.10 The Fisher Exact Test of Independence . . . 185
8.11 Correlation . . . 186
8.11.1 Pearson's r . . . 186
8.11.2 Kendall's τ . . . 189
8.11.3 Spearman's ρ . . . 192
8.11.4 Correlation Is Not Causation . . . 193
Exercises . . . 193
References . . . 194

9 Association and Productivity . . . 197
9.1 Introduction . . . 197
9.2 Cooccurrence Phenomena . . . 198
9.2.1 Collocation . . . 198
9.2.2 Colligation . . . 200
9.2.3 Collostruction . . . 202
9.3 Association Measures . . . 203
9.3.1 Measuring Significant Co-occurrences . . . 203
9.3.2 The Logic of Association Measures . . . 204
9.3.3 A Quick Inventory of Association Measures . . . 205
9.3.4 A Loop for Association Measures . . . 210
9.3.5 There Is No Perfect Association Measure . . . 213
9.3.6 Collostructions . . . 213
9.3.7 Asymmetric Association Measures . . . 222
9.4 Lexical Richness and Productivity . . . 226
9.4.1 Hapax-Based Measures . . . 226
9.4.2 Types, Tokens, and Type-Token Ratio . . . 227
9.4.3 Vocabulary Growth Curves . . . 228
Exercise . . . 235
References . . . 235

10 Clustering Methods . . . 239
10.1 Introduction . . . 239
10.1.1 Multidimensional Data . . . 239
10.1.2 Visualization . . . 240

10.2 Principal Component Analysis . . . 242
10.2.1 Principles of Principal Component Analysis . . . 243
10.2.2 A Case Study: Characterizing Genres with Prosody in Spoken French . . . 243
10.2.3 How PCA Works . . . 245
10.3 An Alternative to PCA: t-SNE . . . 252
10.4 Correspondence Analysis . . . 257
10.4.1 Principles of Correspondence Analysis . . . 257
10.4.2 Case Study: General Extenders in the Speech of English Teenagers . . . 257
10.4.3 How CA Works . . . 261
10.4.4 Supplementary Variables . . . 266
10.5 Multiple Correspondence Analysis . . . 268
10.5.1 Principles of Multiple Correspondence Analysis . . . 269
10.5.2 Case Study: Predeterminer vs. Preadjectival Uses of *Quite* and *Rather* . . . 270
10.5.3 Confidence Ellipses . . . 275
10.5.4 Beyond MCA . . . 276
10.6 Hierarchical Cluster Analysis . . . 276
10.6.1 The Principles of Hierarchical Cluster Analysis . . . 277
10.6.2 Case Study: Clustering English Intensifiers . . . 278
10.6.3 Cluster Classes . . . 279
10.6.4 Standardizing Variables . . . 281
10.7 Networks . . . 283
10.7.1 What Is a Graph? . . . 283
10.7.2 The Linguistic Relevance of Graphs . . . 285
Exercises . . . 290
References . . . 292

A Appendix . . . 295
A.1 Chapter 6 . . . 295
A.1.1 Dispersion Plots . . . 295
A.2 Chapter 8 . . . 297
A.2.1 Contingency Table . . . 297
A.2.2 Discrete Probability Distributions . . . 298
A.2.3 A χ^2 Distribution Table . . . 300

B Bibliography . . . 301

Solutions . . . 309

Index . . . 351

Chapter 1
Introduction

Abstract In this chapter, I explain the theoretical relevance of corpora. I answer three questions: what counts as a corpus?; what do linguists do with the corpus?; what status does the corpus have in the linguist's approach to language?

1.1 From Introspective to Corpus-Informed Judgments

Linguists investigate what it means to know and speak a language. Despite having the same objective, linguists have their theoretical preferences, depending on what they believe is the true essence of language. Supporters of transformational-generative grammar (Chomsky 1957, 1962, 1995) claim that the core of grammar consists of a finite set of abstract, algebraic rules. Because this core is assumed to be common to all the natural languages in the world, it is considered a universal grammar. The idiosyncrasies of language are relegated to the periphery of grammar. Such is the case of the lexicon, context, elements of inter-speaker variation, cultural connotations, mannerisms, non-standard usage, etc.

In generative grammar, pride of place is given to syntax, i.e. the way in which words are combined to form larger constituents such as phrases, clauses, and sentences. Syntax hinges on an opposition between deep structure and surface structure. The deep structure is the abstract syntactic representation of a sentence, whereas the surface structure is the syntactic realization of the sentence as a string of words in speech. For example, the sentence *an angry cow injured a farmer with an axe* has one surface structure (i.e. one realization as an ordered string of words) but two alternative interpretations at the level of the deep structure: (a) an angry cow injured a farmer who had an axe in his hands, and (b) an angry cow used an axe to injure a farmer. In other words, a sentence is generated from the deep structure down to the surface structure.

More generally, generative grammar is a "top-down" approach to language: a limited set of abstract rules "at the top" is enough to generate and account for an infinite number of sentences "at the bottom". What generative linguists truly look for is the finite set of rules that core grammar consists of, in other words speakers' competence, as opposed to speakers' performance. This is to the detriment of idiosyncrasies of all kinds.

Conversely, theories such as functional linguistics (Dik 1978, 1997; Givón 1995), cognitive linguistics (Langacker 1987, 1991; Goldberg 1995), and contemporary typology (Greenberg 1963) advocate a "bottom-up" approach to language. It is usage that shapes the structure of language (Bybee 2006, 2010; Langacker 1988). Grammar is therefore derivative, not generative. There is no point of separating competence and

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_1

performance anymore because competence builds up on performance. In the same vein, grammar has no core or periphery: it is a structured inventory of symbolic units (Goldberg 2003). Therefore, any linguistic unit is worth being taken into consideration with equal attention: morphemes (*un-*, *-ness*), words or phrases (*corpus linguistics*), ritualized or formulaic expressions (*break a leg!*), idioms (*he snuffed it*), non-canonical phrasal expressions (*sight unseen*), semi-schematic expressions (e.g. *just because X doesn't mean Y*), and fully schematic expressions (e.g. the ditransitive construction).

Like biologists, who apprehend life indirectly, i.e. by studying the structure, function, growth, evolution, distribution, and taxonomy of living cells and organisms, linguists apprehend language through its manifestations. For this reason, all linguistic theories rely on native speakers acting as informants to provide data. On the basis of such data, linguists can formulate hypotheses and test them.

Generative linguists are known to rely on introspective judgments as their primary source of data. This is a likely legacy of de Saussure, the father of modern linguistics, who delimited the object of study (*langue*) as a structured system independent from context and typical of an ideal speaker. However, the method can be deemed faulty on at least two accounts. First, there is no guarantee that linguists' introspective acceptability judgments always match what systematically collected data would reveal (Sampson 2001; Wasow and Arnold 2005). Second, for an intuition of well-formedness to be valid, it should at least be formulated by a linguist who is a native speaker of the language under study.[1] If maintained, this constraint limits the scope of a linguist's work and invalidates a significant proportion of the research published worldwide, judging from the number of papers written by linguists who are not fluent speakers of the languages they investigate, including among generativists. This radical position is hardly sustainable in practice. Some generativists working on early language acquisition have used real performance data to better understand the development of linguistic competence.[2]

Because of its emphasis on language use in all its complexity, the "bottom-up" approach provides fertile ground for corpus-informed judgments. It is among its ranks that linguists, dissatisfied with the practice of using themselves as informants, have turned to corpora, judging them to be a far better source than introspection to test their hypotheses. Admittedly, corpora have their limitations. The most frequent criticism that generative linguists level against corpus linguists is that no corpus can ever provide negative evidence. In other words, no corpus can indicate whether a given sentence is impossible.[3] The first response to this critique is simple: grammar rules are generalizations over actual usage and negative evidence is of little import. The second response is that there are statistical methods that can either handle the non-occurrence of a form in a corpus or estimate the probability of occurrence of a yet unseen unit. A more serious criticism is the following: no corpus, however large and balanced, can ever hoped to be representative of a speaker, let alone of a language. Insofar as this criticism has to do with the nature and function of corpora, I address it in Chap. 3.

Linguists make use of corpus-informed judgment because (a) they believe their work is based on a psychologically realistic view of grammar (as opposed to an ideal view of grammar), and (b) such a view can be operationalized via corpus data. These assumptions underlie this book. What remains to be shown is what counts as corpus data.

[1] In fact, Labov (1975) shows that even native speakers do not know how they speak.

[2] See McEnery and Hardie (2012, p. 168) for a list of references.

[3] Initially, generativists claim that the absence of negative evidence from children's linguistic experience is an argument in favor of the innateness of grammatical knowledge. This claim is now used beyond language acquisition against corpus linguistics.

1.2 Looking for Corpus Linguistics

Defining corpus linguistics is no easy task. It typically fills up entire textbooks such as Biber et al. (1998), Kennedy (1998), McEnery and Hardie (2012), and Meyer (2002). Because corpus linguistics is changing fast, I believe a discussion of what it is is more profitable than an elaborate definition that might soon be outdated. The discussion that follows hinges on three questions:

- what counts as a corpus?
- what do linguists do with the corpus?
- what status does the corpus have in the linguist's approach to language?

Each of the above questions has material, technical, and theoretical implications, and offers no straightforward answer.

1.2.1 What Counts as a Corpus

A corpus (plural *corpora*) is a body of material (textual, graphic, audio, and/or video) upon which some analysis is based. Several disciplines make use of corpora: linguistics of course, but also literature, philosophy, art, and science. A corpus is not just a collection of linguistically relevant material. For that collection to count as a corpus, it has to meet a number of criteria: sampling, balance, representativeness, comparability, and naturalness.

1.2.1.1 A Sample of Samples

A corpus is a finite sample of genuine linguistic productions by native speakers. Even a monitor corpus such as The Bank of English, which grows over time, has a fixed size at the moment when the linguist taps into it. Usually, a corpus has several parts characterized by mode (spoken, written), genre (e.g. novel, newspaper, etc.), or period (e.g. 1990–1995), for example. These parts are themselves sampled from a number of theoretically infinite sources.

Sampling should not be seen as a shortcoming. In fact, it is thanks to it that corpus linguistics can be very ambitious. Like astrophysics, which infers knowledge about the properties of the universe and beyond from the study of an infinitesimal portion of it, corpus linguists use finite portions of language use in the hope that they will reveal the laws of a given language, or some aspect of it. A corpus is therefore a sample of samples or, more precisely, a representative and balanced sample of representative and balanced samples.

1.2.1.2 Representativeness

A corpus is representative when its sampling scheme is faithful to the variability that characterizes the target language. Suppose you want to study the French spoken by Parisian children. The corpus you will design for this study will not be representative if it consists only of conversations with peers. To be representative, the corpus will have to include a sizeable portion of conversations with other people, such as parents, school teachers, and other educators.

Biber (1993, p. 244) writes: "[r]epresentativeness refers to the extent to which a sample includes the full range of variability in a population." Variability is a function of situational and linguistic parameters. Situational parameters include mode (spoken vs. written), format (published, unpublished), setting (institutional, public, private, etc.), addressee (present, absent, single, interactive, etc.), author (gender, age, occupation, etc.), factuality (informational, imaginative, etc.), purposes (information, instruction, entertainment, etc.), or topics (religion, politics, education, etc.).

Linguistic parameters focus on the distribution of language-relevant features in a language. Biber (1993, p. 249) lists ten grammatical classes that are used in most variation studies (e.g. nouns, pronouns, prepositions, passives, contractions, or WH-clauses). Each class has a distinctive distribution across text categories, which can be used to guarantee a representative selection of text types. For example, pronouns and contractions are interactive and typically occur in texts with a communicative function. In contrast, WH-clauses require structural elaboration, typical of informative texts.

Distribution is a matter of numerical frequencies. If you are interested in the verbal interaction between waiters and customers in restaurants, unhedged suggestions such as "do you want fries with that?" will be overrepresented if you include 90% of conversations from fast-food restaurants. Conversely, forms of hedged suggestion such as "may I suggest…?" are likely to be underrepresented. To guarantee the representation of the full range of linguistic variation existing in a specific dialect/register/context, distributional considerations such as the number of words per text, the number of texts per text types must be taken into account.

Corpus compilers use sampling methodologies for the inclusion of texts in a corpus. Sampling techniques based on sociological research make it possible to obtain relative proportions of strata in a population thanks to demographically accurate samples. Crowdy (1993) is an interesting illustration of the demographic sampling method used to compile the spoken component of the British National Corpus The British National Corpus (2007). To fully represent the regional variation of British English, the United Kingdom was divided into three supra-regions (the North, the Midlands, and the South) and twelve areas (five for the North, three for the Midlands, and four for the South). Yet, as Biber (1993, p. 247) points out, the demographic representativeness of corpus design is not as important to linguists as the linguistic representativeness.

1.2.1.3 Balance

A corpus is said to be balanced when the proportion of the sampled elements that make it representative corresponds to the proportion of the same elements in the target language. Conversely, an imbalanced corpus introduces skews in the data. Like representativeness, the balance of a corpus also depends on the sampling scheme.

Corpus-linguistics textbooks frequently present the Brown Corpus (Francis and Kučera 1979) and its British counterpart, the Lancaster-Oslo-Bergen corpus (Johansson et al. 1978) as paragons of balanced corpora. Each attempts to provide a representative and balanced collection of written English in the early 1960s. For example, the compilers of the Brown Corpus made sure that all the genres and subgenres of the collection of books and periodicals in the Brown University Library and the Providence Athenaeum were represented. Balance was achieved by choosing the number of text samples to be included in each category. By way of example, because there were about thirteen times as many books in learned and scientific writing as in science fiction, 80 texts of the former genre were included, and only 6 of the latter genre.

Because the Brown Corpus and the LOB corpus are snapshot corpora, targetting a well-delimited mode in a well delimited context, achieving balance was fairly easy. Compiling a reference corpus with spoken data is far more difficult for two reasons. Because obtaining these resources supposes that informants are

willing to record their conversations, they take time to collect, transcribe, and they are more expensive. Secondly, you must have an exact idea of what proportion each mode, genre, or subgenre represents in the target language. If you know that 10% of the speech of Parisian children consists of monologue, your corpus should consist of about 10% of monologue recordings. But as you may have guessed, corpus linguists have no way of knowing these proportions. They are, at best, educated estimates.

1.2.1.4 An Ideal

Sampling methods imply a compromise between what is theoretically desirable and what is feasible. The abovementioned criteria are therefore more of an ideal than an attainable goal (Leech 2007).

Although planned and designed as a representative and balanced corpus, the BNC is far from meeting this ideal. Natural languages are primarily spoken. Yet, 90% of the BNC consists of written texts. However, as Gries (2009) points out, a salient written expression may have a bigger impact on speakers' linguistic systems than a thousand words of conversation. Furthermore, as Biber (1993) points out, linguistic representativeness is far more important than the representativeness of the mode.

1.2.1.5 The Materiality of Corpora

The above paragraphs have shown that a lot of thinking goes on before a corpus is compiled. However, the theoretical status of a corpus should not blind us to the fact that we first apprehend a corpus through the materiality its database, or lack thereof. That materiality is often complex. The original material of a linguistic corpus generally consists of one or several of the following:

- audio recordings;
- audio-video recordings;
- text material.

This material may be stored in a variety of ways. For example, audio components may appear in the forms of reel-to-reel magnetic tapes, audio tapes, or CDs, as was the case back in the days. Nowadays, they are stored as digital files on the web. The datasets of the CHILDES database are available in CHAT, a standardised encoding format. The files, which are time aligned and audio linked, are to be processed with CLAN (`http://childes.psy.cmu.edu/`). In general, multimodal components include both the original audio/video files and, optionally, a written transcription to allow for quantification.

As a corpus linguist working mainly on the English language, I cannot help paying tribute to the first "large-scale" corpus of English: the Survey of English Usage corpus (also known as the Survey Corpus). It was initiated by Randolph Quirk in 1959 and took thirty years to complete. The written component consists of 200 texts of 5000 words each, amounting to one million words of British English produced between 1955 and 1985. The Survey Corpus also has a spoken component in the form of now digitized tape recordings of monologues and dialogues. The corpus is originally compiled from magnetic data recordings, transcribed by hand, and typed on thousands of 6-by-4-inch paper slips stored in a filing cabinet. Each lexical item is annotated for grammatical features and stored likewise. For examples, all verb phrases are filed in the verb phrase section. The spoken component is also transcribed and annotated for prosodic features. It still exists under the name of the London-Lund Corpus.

The Survey Corpus was not computerized until the early 1980s.[4] With a constant decrease in the price of disk storage and an ever increasing offer of tools to process an ever increasing amount of electronic data, all corpus projects are now assumed to be digital from stage one. For this reason, when my students fail to bring their laptop to their first corpus linguistics class, I suggest they take a trip to University College London and browse manually through the many thousands of slips filed in the cabinets of the Survey Corpus. Although the students are always admirative of the work of corpus linguistics pioneers (Randoph Quirk, Sidney Greenbaum, Geoffrey Leech, Jan Svartvik, David Crystal, etc.), they never forget their laptops afterwards.

1.2.1.6 Does Size Matter?

Most contemporary corpora in the form of digital texts (from either written or transcribed spoken sources) are very large. The size of a corpus is relative. When it first came out, in the mid 1990s, the BNC was considered a very big corpus as it consisted of 100 million words. Compared to the Corpus of Contemporary American English (450 million words), the Bank of English (45 billion words), the ukWaC corpus of English (2.25 billion words),[5] or Sketch Engine's enTenTen12 corpus (13 billion words), the BNC does not seem that large anymore.

What with the availability of digital material and a rich offer of automatic annotation and markup tools, we are currently witnessing an arms race in terms of corpus size. This is evidenced by the increasing number of corpora that are compiled from the web (Baroni et al. 2009). On the one hand, a large corpus guarantees the presence of rare linguistic forms. This is no trifling matter insofar as the distribution of linguistic units in language follows a Zipfian distribution: there is a large number of rare units. On the other hand, a very large corpus loses in terms of representativeness and balance, unless huge means are deployed to compile it with respect to the abovementioned corpus ideal (which, in most cases, has an unbearable cost). Finally, no corpus extraction goes without noise (i.e. unwanted data that is hard to filter out in a query). The larger the corpus, the more the noise.

Depending on your case study and the language you investigate, using a small corpus is not a bad thing. For example, Ghadessy et al. (2001) show that small corpora can go a long way in English language teaching. Unless you study a macrolanguage (e.g. Mandarin, Spanish, English, or Hindi), it is likely that you will not be able to find any corpus. The problem is even more acute for endangered or minority languages (e.g. Breton in France, or Amerin languages in the Americas) for which you only have a sparse collection of texts and sometimes punctual recordings at your disposal. Once compiled into a corpus, those scarse resources can serve as the basis of major findings, such as Boas and Schuchard (2012) in Texas German, or Hollmann and Siewierska (2007) in Lancashire dialects.

Small size becomes a problem if the unit you are interested in is not well represented. All in all, size matters, but if it is wisely used, a small corpus is worth more than a big corpus that is used unwisely.

1.2.2 What Linguists Do with the Corpus

Having a corpus at one's disposal is a necessary but insufficient condition for corpus linguistics. Outside linguistics, other disciplines in the humanities make use of large-scale collections of texts. What they gener-

[4] Transcriptions of the spoken component of the Survey Corpus were digitized in the form of the London-Lund Corpus.

[5] Ferraresi (2007).

ally do with those texts is known as text mining. Corpus-linguists may use text-mining techniques at some point (for instance when they make frequency lists). However, the goal of text mining techniques is to obtain knowledge about a text or group of texts (Jockers 2014), whereas the goal of corpus-linguistics techniques is to better understand the rules governing a language as a whole, or at least some aspect of that language (e.g. a specific register or dialect).

1.2.2.1 Generalization

You do not suddenly become a corpus linguist by running a query in a digital text database in search of a savory example.[6] You investigate a corpus because you believe your findings can be extended to the target language or variety. This is called generalization.

Generalization implies a leap of faith from what we can infer from a finite set of observable data to the laws of the target language. Outside corpus linguistics, not everyone agrees. In an interview (Andor 2004, p. 97), Chomsky states that collecting huge amounts of data in the hope of coming up with generalizations is unique in science. He makes it sound like it is a pipe dream. To reformulate Chomsky's claim, a corpus should not be a basis for linguistic studies if it cannot represent language in its full richness and complexity. Most corpus linguists rightly counter that they do not aim to explain all of a language in every study (Glynn 2010) and that the limits of their generalizations are the limits of the corpus. Furthermore, even if no corpus can provide access to the true, unknown law of a language, a corpus can be considered a sample drawn from this law. As you will discover in the second part of the book, there are ways of bridging the gap between what we can observe and measure from a corpus, and what we do not know about a language. Ambitious statistics is needed to achieve this. I am specifically referring to the statistics used in biostatistics, where scientists commonly bridge the gap between what they can observe from a group of patients and what they can infer about a disease, contra Chomsky's intuition.

1.2.2.2 Quantification

Corpus linguistics is quantitative when the study of linguistic phenomena based on corpora is systematic and exhaustive. Gries (2014, p. 365) argues that corpus linguistics in the usage-based sense of the word is a distributional science. A distributional science infers knowledge from the distribution, dispersion, and co-occurrence of data. For this reason, quantitative corpus linguists typically focus on corpus frequencies by means of frequency lists, concordances, measures of dispersion, and co-occurrence frequencies.

All linguists aim at some form of generalization, but not all of them engage in some form of quantification to meet this aim. This means that you can make a qualitative and quantitative use of a corpus. Qualitative corpus analysis may consist in formulating a hypothesis and testing it by looking for potential counterexamples in the corpus. It may also consist in using the corpus to refine the hypothesis at some stage. The concordancer is a corpus tool par excellence meant for qualitative inspection. It is used to examine a word in its co-text regardless of any form of quantification.

An oft-heard misconception about corpus linguistics is that the quantitative methods involved are suspiciously too objective and miss some information that only the linguist's expert subjectivity can bring. Nothing is further from the truth. First, you should never start exploring a corpus, let alone quantify your findings, if you do not have a research question, a research hypothesis, and a clear idea as to how the corpus

[6] The practise is known as "chasing butterflies".

will help you answer the question and test the hypothesis. Second, you should never consider the quantified corpus findings as the final stage in your research project. Uninterpreted results are useless because they do not speak for themselves: "quantification in empirical research is not about quantification, but about data management and hypothesis testing" (Geeraerts 2010).

1.2.3 How Central the Corpus Is to a Linguist's Work

Some linguists believe that using a corpus is but one of several steps in a research project (although a significant one). For example, a linguist might generate a working hypothesis regarding a linguistic form and then decide to test that hypothesis by observing the behavior of that linguistic form in a corpus and by generalizing over the findings. Other linguists adopt a more radical stance. They claim that the corpus is the only possible approximation we have of a speaker's linguistic competence. It is therefore a linguist's job to investigate corpora.

1.2.3.1 The Corpus as a Step in the Empirical Cycle

To most corpus linguists, corpora and quantitative methods are only a moment in what the cognitive semanticist Dirk Geeraerts calls "the empirical cycle" (Geeraerts 2010).[7] Empirical data are of two kinds: observational and experimental. Observational data are collected as they exist. In contrast, elicited data have to be obtained experimentally. Corpus data are observational, as opposed to elicited data, which are experimental.

The flowchart in Fig. 1.1 is a representation of D. Geeraerts's empirical cycle, adapted to quantitative corpus linguistics. The corpus, which appears in green is but a moment, albeit a central one, of the empirical cycle. The left part of the chart (where the steps are connected by means of dashed arrows) is iterative. The cycle involves several rounds of hypothesis formulating/updating, operationalizing, corpus data gathering, and hypothesis testing. If the empirical testing is satisfactory, the findings can inform back theory (see the right part of the chart), which in turn helps formulate other hypotheses, and so on and so forth.

Those who reject corpus linguistics on the basis that it is too objective should notice that nearly all the blocks of the flowchart involve subjective appreciation from the linguist. This is valid also for the block "test hypotheses" with quantification and statistics insofar as you cannot quantify or run statistics blindly. The choice of quantitative methods depends largely on what the corpus gives you.

1.2.3.2 The Corpus as the Alpha and Omega of Linguistics

Some linguists adopt a much stronger stance on the place of corpora in their work. They consider that the grammar of speakers takes the form of a mental corpus.

The idea of grammar being represented as a mental corpus originates from outside corpus linguistics per se. Cognitive Grammar, an influential usage-based theory of language, defines grammar as a structured inventory of symbolic linguistic units (Langacker 1987, p. 57). In this theory, grammar is the psychological representation of linguistic knowledge. A unit is symbolic because its formal component and its semantic

[7] In case it is not clear enough, this paper is a must-read if you plan to do corpus-based semantics.

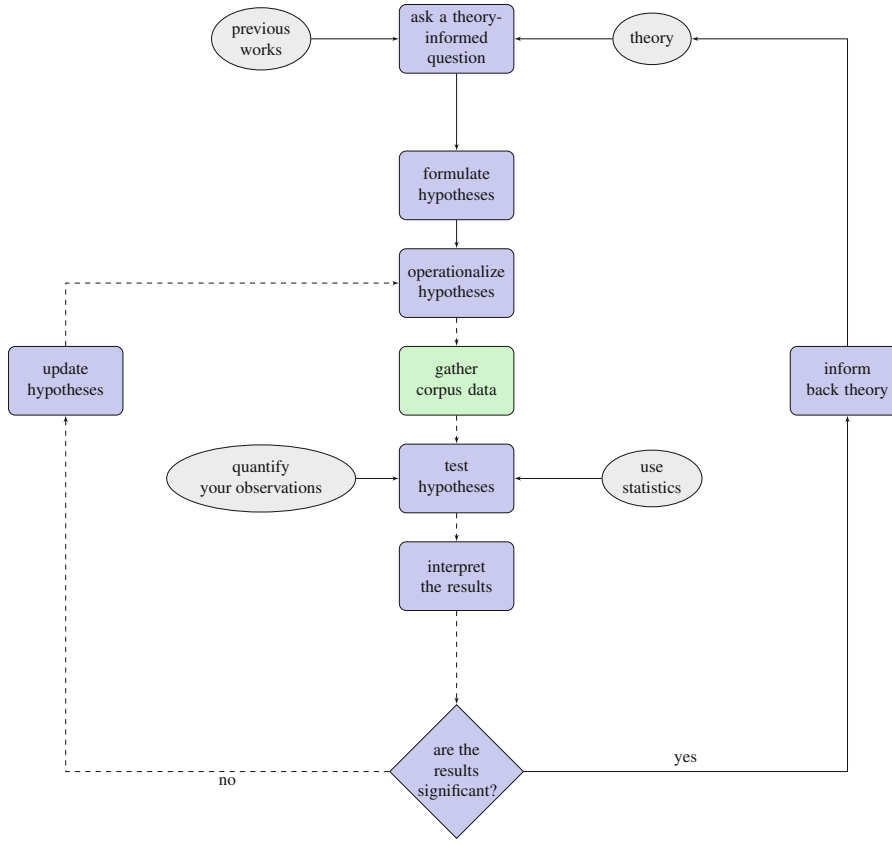


Fig. 1.1: The empirical cycle (adapted to quantitative corpus linguistics)

component are linked by conventions. This is reminiscent of Ferdinand de Saussure's "arbitrariness of the sign". Unlike generative linguistics, which separates form and meaning, Cognitive Grammar advocates a uniform representation that combines both. Central in Cognitive Grammar, and of major interest to corpus linguistics because of its roots in frequency, is the concept of entrenchment: "[w]ith repeated use, a novel structure becomes progressively entrenched, to the point of becoming a unit [...]" (Langacker 1987, p. 59). Each time a unit (a phoneme, a morpheme, a morphosyntactic pattern, etc.) is used, it activates a node or a pattern of nodes in the mind. The more the unit occurs, the more likely it is to be stored independently. In other words, linguistic knowledge is a repository of units memorized from experiences of language use. Each experience leaves a trace in the speaker's memory and interacts with previously stored units. This dynamic repository is what Taylor (2012) calls a mental corpus.

In line with the above, some linguists consider that the closest approximation of a mental corpus is a linguist's corpus. Tognini-Bonelli (2001) distinguishes between *corpus-based* and *corpus-driven* linguistics. Corpus-based linguistics is the kind exemplified in Sect. 1.2.3.1. Corpus-driven linguistics is the radical extension of corpus-based linguistics: the corpus is not part of a method but our sole access to language competence. According to McEnery and Hardie (2012), this idea has its roots in the works of scholars

inspired by J.R. Firth, such as John Sinclair, Susan Hunston, Bill Louw, Michael Stubbs, or Wolfgang Teubert. Studies in this tradition typically focus on collocations and discourse effects. Let me zoom in on collocation, as it is a concept we shall come back to later.

According to Firth (1957), central aspects of the meaning of a linguistic unit are lost if we examine the unit independently from its context.[8] For example, a seemingly vague word such as *something* acquires a negative meaning once it precedes the verb *happen*, as in: *[...] a lady stopped me and she said has something happened to your mother? And I said, yes, she died [...]* (BNC-FLC). A unit and its context are not mere juxtapositions. They form a consistent network whose nodes are bound by mutual expectations (Firth 1957, p. 181). Firth was not concerned with corpora but by the psychological relevance of retrieving meaning from context. Firth's intuitions were translated into a corpus-driven methodology through the work of "neo-Firthians", such as Sinclair (1966, 1987, 1991), Sinclair and Carter (2004), Teubert (2005), or Stubbs (2001).

As always, even though radical ideas are often theoretically challenging, the middle ground is preferable in practice. This is why most quantitative corpus linguists consider that a corpus is a central part of research, but not the alpha and omega of linguistics. You can still do corpus-based linguistics and make use of popular corpus-driven methods such as concordances or association measures.

References

Andor, József. 2004. The Master and His Performance: An Interview with Noam Chomsky. *Intercultural Pragmatics* 1 (1): 93–111. doi:10.1515/iprg.2004.009.

Baroni, Marco, et al. 2009. The WaCky Wide Web: A Collection of Very Large Linguistically Processed Web-Crawled Corpora. *Language Resources and Evaluation* 43 (3): 209–226.

Biber, Douglas. 1993. Representativeness in Corpus Design. *Literary and Linguistic Computing* 8 (4): 241–257.

Biber, Douglas, Susan Conrad, and Randi Reppen. 1998. *Corpus Linguistics: Investigating Language Structure and Use*. Cambridge: Cambridge University Press.

Boas, Hans C., and Sarah Schuchard. 2012. A Corpus-Based Analysis of Preterite Usage in Texas German. In *Proceedings of the 34th Annual Meeting of the Berkeley Linguistics Society*.

Bybee, Joan. 2010. *Language, Usage, and Cognition*. Cambridge: Cambridge University Press.

Bybee, Joan L. 2006. From Usage to Grammar: The Mind's Response to Repetition. *Language* 82 (4): 711–733.

Chomsky, Noam. 1957. *Syntactic structures*. The Hague: Mouton.

Chomsky, Noam. 1962. *Aspects of the Theory of Syntax*. Cambridge, MA: MIT Press.

Chomsky, Noam. 1995. *The Minimalist Program*. Cambridge, MA: MIT Press.

Crowdy, Steve. 1993. Spoken Corpus Design. *Literary and Linguistic Computing* 8 (4): 259–265.

Dik, Simon C. 1978. *Functional Grammar*. Amsterdam, Oxford: North-Holland Publishing Co.

Dik, Simon C. 1997. *The Theory of Functional Grammar*, ed. Kees Hengeveld, 2nd ed. Berlin, New York: Mouton de Gruyter.

Ferraresi, Adriano. 2007. Building a Very Large Corpus of English Obtained by Web Crawling: ukWaC. Master's Thesis. University of Bologna.

[8] "You shall know a word by the company it keeps" (Firth 1957, p. 179).

Firth, John. 1957. A Synopsis of Linguistic Theory, 1930–1955. In *Selected Papers of J.R. Firth 1952–1959*, ed. Frank Palmer, 168–205. London: Longman.

Francis, W. Nelson, and Henry Kučera. 1979. *Manual of Information to Accompany a Standard Corpus of Present-Day Edited American English, for Use with Digital Computers*. Department of Linguistics. Brown University. http://www.hit.uib.no/icame/brown/bcm.html(visited on 03/10/2015).

Geeraerts, Dirk. 2010. The Doctor and the Semantician. In *Quantitative Methods in Cognitive Semantics: Corpus-Driven Approaches*, ed. Dylan Glynn and Kerstin Fischer, 63–78. Berlin, Boston: Mouton De Gruyter.

Ghadessy, Mohsen, Alex Henry, and Robert L. Roseberry. 2001. *Small Corpus Studies and ELT*. Amsterdam: John Benjamins.

Givón, Talmy. 1995. *Functionalism and Grammar*. Amsterdam: John Benjamins.

Glynn, Dylan. 2010. Corpus-Driven Cognitive Semantics: Introduction to the Field. In *Quantitative Methods in Cognitive Semantics: Corpus-driven Approaches*, 1–42. Berlin: Mouton de Gruyter.

Goldberg, Adele E. 1995. *Constructions: A Construction Grammar Approach to Argument Structure*. Chicago: University of Chicago Press.

Goldberg, Adele E. 2003. Constructions: A New Theoretical Approach to Language. *Trends in Cognitive Sciences* 7 (5): 219–224. http://dx.doi.org/10.1016/S1364-6613(03)00080-9.

Greenberg, Joseph H. 1963. Some Universals of Grammar with Particular Reference to the Order of Meaningful Elements. In *Universals of Human Language*, ed. Joseph H. Greenberg, 73–113. Cambridge: MIT Press.

Gries, Stefan Thomas. 2009. What is Corpus Linguistics? *Language and Linguistics Compass* 3: 1225–1241. doi:10.1111/j.1749-818X.2009.00149.x.

Gries, Stefan Thomas. 2014. Frequency Tables: Tests, Effect Sizes, and Explorations. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 365–389. Amsterdam: John Benjamins.

Hollmann, Willem B., and Anna Siewierska. 2007. A Construction Grammar Account of Possessive Constructions in Lancashire Dialect: Some Advantages and Challenges. *English Language and Linguistics* 11 (2): 407–424. doi:10.1017/S1360674307002304.

Jockers, Matthew. 2014. *Text Analysis with R for Students of Literature*. New York: Springer.

Johansson, Stig, Geoffrey Leech, and Helen Goodluck. 1978. *Manual of Information to Accompany the Lancaster-Oslo/Bergen Corpus of British English, for Use with Digital Computers*. Department of English. University of Oslo. http://clu.uni.no/icame/manuals/LOB/INDEX.HTM (visited on 03/10/2015).

Kennedy, Graeme. 1998. *An Introduction to Corpus Linguistics*. Harlow: Longman.

Labov, William. 1975. Empirical Foundations of Linguistic Theory. In *The Scope of American Linguistics: Papers of the First Golden Anniversary Symposium of the Linguistic Society of America*, ed. Robert Austerlitz, 77–133. Lisse: Peter de Ridder.

Langacker, Ronald W. 1987. *Foundations of Cognitive Grammar: Theoretical Prerequisites*, Vol. 1. Stanford: Stanford University Press.

Langacker, Ronald W. 1988. A Usage-Based Model. In *Topics in Cognitive Linguistics*, ed. Brygida Rudzka-Ostyn, 127–161. Amsterdam, Philadelphia: John Benjamins.

Langacker, Ronald W. 1991. *Foundations of Cognitive Grammar: Descriptive Application*, Vol. 2. Stanford: Stanford University Press.

Leech, Geoffrey. 2007. New Resources, or Just Better Old Ones? The Holy Grail of Representativeness. In *Corpus Linguistics and the Web*, ed. Marianne Hundt, Nadja Nesselhauf, and Carolin Biewer, 133–149. Amsterdam: Rodopi.

McEnery, Tony, and Andrew Hardie. 2012. *Corpus Linguistics : Method, Theory and Practice*. Cambridge Textbooks in Linguistics. Cambridge, New York: Cambridge University Press.

Meyer, Charles F. 2002. *English Corpus Linguistics: An Introduction*. Cambridge: Cambridge University Press.

Sampson, Geoffrey. 2001. *Empirical Linguistics*. London: Continuum.

Sinclair, John. 1966. Beginning the Study of Lexis. In *In Memory of J.R. Firth*, ed. C.E. Bazell, et al., 410–431. London: Longman.

Sinclair, John. 1987. Collocation: A Progress Report. In *Language Topics: Essays in Honour of Michael Halliday*, ed. R. Steele, and T. Threadgold, Vol. 2, 319–331. Amsterdam: John Benjamins.

Sinclair, John. 1991. *Corpus, Concordance, Collocation*. Oxford: Oxford University Press.

Sinclair, John, and Ronald Carter. 2004. *Trust the Text: Language, Corpus and Discourse*. London: Routledge

Stubbs, Michael. 2001. *Words and Phrases: Corpus Studies of Lexical Semantics*. Oxford: Wiley–Blackwell.

Taylor, John R. 2012. *The Mental Corpus: How Language is Represented in the Mind*. Oxford: Oxford University Press.

Teubert, Wolfgang. 2005. My Version of Corpus Linguistics. *International Journal of Corpus Linguistics* 10 (1): 1–13. doi:10.1075/ijcl.10.1.01teu.

The British National Corpus. 2007. *BNC XML Edition*. Version 3. http://www.natcorp.ox.ac.uk/. Distributed by Oxford University Computing Services On Behalf of the BNC Consortium.

Tognini-Bonelli, Elena. 2001. *Corpus Linguistics at Work*. Amsterdam: John Benjamins.

Wasow, Thomas, and Jennifer Arnold. 2005. Intuitions in Linguistic Argumentation. *Lingua* 115 (11): 1481–1496. http://dx.doi.org/10.1016/j.lingua.2004.07.001.

Part I
Methods in Corpus Linguistics

If you are an absolute beginner, there is a chance that coding will be frustrating at first. The good news is that it will not last. An excellent reason for being optimistic is that you will never code alone. R is maintained, developed, and supported by a big community of friendly users. Before you know it, you will be part of that community and you will be writing your own R scripts. You will find below a list of helpful online ressources.

Getting help from the online community

- Stack Overflow: a question and answer site on a variety of topics, including R
  `http://stackoverflow.com/questions/tagged/r`
- R-bloggers: a compilation of bloggers' contributions on R
  `http://www.r-bloggers.com/`
- inside-R: a community site dedicated to R
  `http://www.inside-r.org/blogs`
- GitHub: a collaborative code repository (not limited to R)
  `https://github.com/`
  type `#R` in GitHub's search engine
- The R mailing list
  `https://stat.ethz.ch/mailman/listinfo/r-help`

The most frequently asked question from my students is "how long does it take to master R?". There is no straight answer because several variables are involved.

First of all, you do not need to master all of R's features to start using it in corpus linguistics. As you will realize in the next chapter, generating a frequency list does not require an engineer's skills.

Secondly, even though your programming experience does make a difference in how fast you learn the R language, it does not teach you how to select the appropriate tool for the appropriate task. This means that your learning of the R language is a direct function of what you need to do with it. The best R learners among my students also happen to be the best linguists.

Thirdly, over and above your programming experience, your level of motivation is key. Every time I teach a course on corpus linguistics and statistics with R, it is only a matter of days until I receive my first emails from students asking me for advice regarding some advanced task that we have not covered yet and whose code they are eager to crack.

You can take advantage of this book in two ways. The most basic use is to treat it like a recipe book because you do not want to invest time in learning the inner logic of a programming language. In this case, you just need to adapt the scripts to suit your needs. A more advanced (and far more beneficial) use is to take the book as an opportunity to learn corpus techniques along with a new programming language, killing two birds with one stone. From my experience, the asset of the second option is that learning R will help you become a better quantitative corpus linguist. In any case, just like learning how to play the piano, it is only through regular practice that you will get familiar with R commands, R objects, R scripts, functions, and even learn to recognize R oddities (Burns 2011).

Chapter 2
R Fundamentals

Abstract This chapter is designed to get linguists familiar with the R environment. First, it explains how to download and install R and R packages. It moves on to teach how to enter simple commands, use ready-made functions, and write user-defined functions. Finally, it introduces basic R objects: the vector, the list, the matrix, and the data frame. Although meant for R beginners, this chapter can be read as a refresher course by those readers who have some experience in R.

2.1 Introduction

If you are new to R, this chapter is the place to start. If you have some experience with R or if you are too impatient, feel free to skip those pages. You can still come back to them if you get stuck at some point and hit the wall.

There are many other books that do a great job at introducing R in depth. This chapter is not meant to compete with them. Nevertheless, because I firmly believe that getting familiar with R basics is a prerequisite to understand what we can truly do as corpus linguists, I have designed this chapter to provide an introduction to R that both covers the bare necessities and anticipates the kind of coding and data structures involved in corpus linguistics.

2.2 Downloads and Installs

R is an interpreted language: the user types and enters the commands from a script and the system interprets the commands. Most users run R from a window system: the commands are entered via the R console.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_2

2.2.1 Downloading and Installing R

First, visit the Comprehensive R Archive Network (CRAN, https://cran.r-project.org/) and download the most recent release of R.[1]

- If you are a Windows user, click on "Download R for Windows", then on "base". Download the setup file (`.exe`) by clicking on "Download R 3.X.X for Windows" (where 3.X.X stands for the identification number of the most recent release) and run it. The installation program gives you step by step instructions to install R in the default directory of your system. R should appear in the usual *Programs* folder. Double click the icon to launch the R graphical user interface (GUI).
- If, like me, you are a Mac OS or macOS user, click on "Download R for (Mac) OS X" and select the release that corresponds to your OS version (i.e. Snow Leopard, Lion, Moutain Lion, Mavericks, or Yosemite). The file takes the form of an Apple software package whose name ends with `.pkg`. The installer guides you until R is installed on your system. You should find it in the usual *Applications* folder. Double click the icon to launch the R GUI.
- Finally, if you are a Linux user, click on "Download R for Linux", open the parent directory that corresponds to your distribution (Debian, Red Hat, SUSE, or Ubuntu), and follow the instructions until you have the base installation of R on your system. To start R, type "R" at the command line.

You now have the base installation of R on your system. You may want to visit the CRAN website once in a while to make sure you have the most updated release, which includes bugfixes. However, before you upgrade, I advise you to make sure that all your packages (see Sect. 2.5 below) are compatible with that version. Most of the time, packages are tested on beta versions of the most recent R release, but it pretty much depends on how much time the authors have until the official version is released. Note that you can always easily come back to older releases if incompatibility issues arise.

2.2.2 Downloading and Installing RStudio

RStudio is an Integrated Development Environment (IDE) for R. In other words, R must be installed first for RStudio to run. RStudio has many added values, the main one in my view being a nice windowing display that allows you to visualize the R console, scripts, plots, data, packages, files, etc. in the same environment.

Although I appreciate RStudio, as long as it remains free, open source, and faithful to the initial spirit of those who created R, my own preference goes to the basic console version of R. I do not have any objective reason for this, except that I did my first steps in R using the R console. My workflow is now based on the console. In the remainder of the book, all instructions and illustrations will be relative to the console version of R and entirely compatible with RStudio.

To download and install RStudio Desktop (Open Source Edition), visit https://www.rstudio.com/ and follow the instructions.

[1] At the time of writing, the latest R release is 3.3.1, "Bug in Your Hair".

2.2.3 Downloading the Book Materials

Now that R and/or RStudio are installed, it is time to download the book materials. You can download a zipped archive (`CLSR.zip` for "Corpus Linguistics and Statistics with R") from http://extras.springer.com/2018/978-3-319-64570-4. The archive contains:

- the code for each chapter;
- input files.

Unzip the archive and place the resulting `CLSR` folder at the root of your hard drive, i.e. `C:` on Windows and `Macintosh HD` on a Mac. One obvious reason for this location is that you will not have to type long pathnames when you want to access the files in the folder.

2.3 Setting the Working Directory

Before proceeding further, you must set the working directory. The working directory is a folder which you want R to read data from and store output into. Because you are probably using this book as a companion in your first foray into quantitative linguistics with R, I recommend that you set the `CLSR` folder as your working directory.

You are now about to enter your first R command. Commands are typed directly in the R console, right after the prompt `>`. To know your default working directory, type `getwd()` and press ENTER.

```
> getwd()
```

Set the working directory by entering the path to the desired directory/folder:

```
> setwd("C:/CLSR") # Windows
> setwd("/CLSR") # Mac
```

Finally, make sure the working directory has been set correctly by typing `getwd()` again.[2]

2.4 R Scripts

Simple operations can be typed and entered directly into the R console. However, corpus linguistics generally involves a series of distinct operations whose retyping is tedious. To save you the time and trouble of retyping the same lines of code and to separate commands from results, R users type commands in a script and save the script in a file with a special extension (`.r` or `.R`). Thanks to this extension, your system knows that the file must be opened in R.

To create a script file, there are several options:

- via the drop-down menu: `File > New Document`;
- via the R GUI: click on the blank page;

[2] Note that if you have a PC running on Windows, you might be denied access to the `C:` drive. The default behavior of that drive can be overridden.

- via a text editor[3]: create a new text file and save it using the `.r` or `.R` extension.

R users store their scripts in a personal library for later use so that they can reuse whole or bits of scripts for new tasks. I cannot but encourage you to do the same.

2.5 Packages

R comes equipped with pre-installed packages. Packages are external community-developed extensions in the form of libraries. They add functionalities not included in the base installation such as extra statistical techniques, extended graphical possibilities, data sets, etc. As of November 2016, the CRAN package repository displays 9442 available packages.[4] Like R releases, package versions are regularly updated. Over time, some packages may become deprecated, and R will let you know with a warning message.

2.5.1 Downloading Packages

Packages can be downloaded in several ways. One obvious way is to use the drop-down menu from the R GUI: *Packages & Data* > *Package Installer*. Another way is to enter the following:

```
> install.packages()
```

When you download a package for the first time, R prompts you to select a mirror, i.e. a CRAN-certified server, by choosing from a list of countries. Select the country that is the closest to you. R then displays the list of all available packages. Click on the one(s) you need, e.g. `ggplot2`. Alternatively, if you know the name of the package you need, just enter:

```
> install.packages("ggplot2") # make sure you do not forget the quotes!
```

Finally, if you want to install several packages at the same time, type the name of each package between quotes, and separate the names with a comma inside `c()`:

```
install.packages(c("Hmisc", "FactoMineR"))
```

Note that R will sometimes install additional packages so that your desired package runs properly. These helping packages are known as dependencies. You do not need to install your packages again when you open a new R session.

In RStudio, the above works, but you may also want to use the drop-down menu: *Tools* > *Install Packages* in the pop-up window, type the package name under *Packages* and check the box *Install dependencies*.

[3] Free text editors abound. I recommend Notepad++ (https://notepad-plus-plus.org/) or Tinn-R (http://www.sciviews.org/Tinn-R/) for Windows users, and TextWrangler (http://www.barebones.com/products/textwrangler/) for Mac users.

[4] See https://cran.r-project.org/web/packages/ for an updated count.

2.5.2 Loading Packages

Downloading a package means telling R to store it in its default library. The package is still inactive.

```
> find.package("ggplot2")
[1] "/Library/Frameworks/R.framework/Versions/3.3/Resources/library/ggplot2"
```

You cannot use a package unless you load it onto R. This is how you do it:

```
> library(ggplot2) # this time, you do not need the quotes!
```

You will need to load the desired package(s) every time you start a new R session. Although this may seem tedious at first, it is in fact a good thing. After spending some time using R, you will have accumulated an important collection of packages, whose size would make R much slower if all were always active.

2.6 Simple Commands

The simplest task that R can do is probably a simple arithmetic calculation such as an addition,

```
> 3+2
[1] 5
```

a subtraction,

```
> 3-2
[1] 1
```

a multiplication,

```
> 3*2 # note the use of the asterisk *
[1] 6
```

a division,

```
> 3/2
[1] 1.5
```

or an exponentiation.

```
> 3^2 # 3 to the power of 2, note the use of the caret ^
[1] 9
```

The second line is R's answer. The comment sign # tells R not to interpret what follows as code. It is very convenient for commenting your code, as I have done twice above. The number between square brackets before the result is R's way of indexing the output. In the above example, the output consists of only one element. When the result is long, indexing becomes more useful:

```
> 1:30 # a sequence of numbers from 1 to 30 by increment of 1.
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

Here, `[1]` indicates that 1 is the first element of the output, `[16]` that 16 is the sixteenth element.

Like any pocket calculator, R gives priority to multiplications and divisions over additions and subtractions. Compare:

```
> 10*3-2
[1] 28
```

and

```
> 10*(3-2)
[1] 10
```

Interestingly, R has some built-in values and mathematical functions, many of which we will return to:

```
> abs(-13) # absolute value
[1] 13
> pi # the ratio of the circumference of a circle to its diameter
[1] 3.141593
> sqrt(8) # square root
[1] 2.828427
> round(7.23125)
[1] 7
```

You can nest these built-in functions:

```
> round(sqrt(pi))
[1] 2
```

As will soon appear, R is not just a powerful calculator. It is equally good at many other things such as processing text, data structures, generating elaborate graphs, doing statistics, etc.

2.7 Variables and Assignment

Without a way of storing intermediate results, no programming language could work. This is why all programming languages use variables. A variable is a named data structure stored in the computer's working memory. It consists of alphanumeric code to which some programming data is assigned. For instance, you may store someone's age (e.g. 40) in a variable named `age`.

```
> age <- 40
```

From now on, each type you type `age`, the language interprets the variable as standing for its value, i.e. 40.

```
> age
[1] 40
```

Like all programming languages, R stores programming data in variables via assignment. In the example below, R assigns the numeric value 3 to the variable `a` and the numeric value 2 to the variable `b` thanks to `<-`. The assignment operator `<-` is a combination of the "less than" symbol followed by the hyphen with no space between them. The result is a left-facing arrow.[5]

```
> a <- 3 ; a
[1] 3
> b <- 2 ; b
[1] 2
```

[5] Even though I do not use it in this book to avoid confusion, the equal sign `=` can also be used for assignment instead of `<-`. Note that you can separate two or more commands with a semi-colon `;`.

From now on, each variable stands for and behaves like its value.

```
> a+1
[1] 4
> a/b
[1] 1.5
```

Variables must follow a specific syntax. You are free to choose the variable name, providing there is no space in it. For example, `example_sum` is ok, but `example sum` is not. Also, you had better keep the variable names short, to save time in case you need to retype them later. R is case-sensitive, which means the variable `a` is different from the variable `A`. Regarding the assignment operator, it does not matter if you place a space before and after `<-`. However, `<` and `-` should never be separated with a space.

2.8 Functions and Arguments

In R, you can use ready-made functions or create your own user-defined functions. Let us start with ready-made functions.

2.8.1 Ready-Made Functions

Functions are preset instructions to R. You recognize a function by its name followed by a pair of brackets, e.g. `somefunctionname()`. A function takes arguments, i.e. elements to which the instruction will be applied and specifications as to how to apply these instructions. Arguments appear between the brackets of the function. How many and what kinds of arguments a function takes depends on the function. To access that information, just type `help()` and the name of the function in the brackets or just a question mark `?` followed by the function name with no space and no brackets (e.g. `?mean`).[6] All functions validated by CRAN have a help page.

In the example below, `mean` is a function for the arithmetic mean. When you type `?mean`, R opens a help window for the function. You see that it minimally takes three arguments expressed in the form `mean(x, trim = 0, na.rm = FALSE, ...)`. The first argument (`x`) is an R object such as a numeric vector of *n* observations. The second argument (`trim`) is "the fraction (0–0.5) of observations to be trimmed from each end of `x` before the mean is computed". The third argument (`na.rm`) is "a logical value indicating whether `NA` values should be stripped before the computation proceeds". In the example below, we ask R to compute the arithmetic mean of a sequence of numbers from 1 to 20 incremented by one, with no trimming and no removal of `NA` values (since there aren't any). Arguments are specified explicitly:

```
> mean(1:20, trim = 0, na.rm = FALSE)
[1] 10.5
```

When the argument names are unspecified, R processes the arguments in the default order it expects them to appear:

```
> mean(1:20, 0, FALSE)
[1] 10.5
```

[6] When you type the function name and the first bracket, R recognizes the function and displays the list of all possible arguments at the bottom of the console.

Here, R understands that the first argument is numeric, the second argument is the value of `trim`, and the third argument is the logical value of `na.rm`. As seen above, the default value of `trim` is 0 and the default value of `na.rm` is `FALSE`. Since we do not change the default settings of these last two arguments, we do not even need to include them:

```
> mean(1:20)
[1] 10.5
```

2.8.2 User-Defined Functions

In R, you can create your own user-defined function. A function involves three steps:

- defining the function;
- loading the function;
- running the function.

You may add a fourth step: fine-tuning the function.

2.8.2.1 Defining the Function

All functions follow this format:

```
> function_name <- function(arg1, arg2, ... ){
+   statements # some code
+   return(object) # the value returned by the function, i.e. the result
+ }
```

Here, `function_name` is the name of your function, which you are free to choose as long as it is not the name of a preset function (e.g. `sum`, `function`, `mean`, `plot`, etc.). The function `function_name()` takes arguments, here `arg1, arg2`. Use as many arguments as your function needs to run. You can specify a default value for a given argument. For instance, `arg3=pi` would stipulate that the third argument takes the default value 3.141593 unless specified otherwise. If you do not specify a default value, R expects you to specify one when you run the function. The dots `(...)` allow for other arguments to be passed to your function or from other functions or methods. For example, it is a convenient way of embedding other functions in your own function. The curly brackets ({ }) delimit the code that the body of your function consists of. Finally, the last line of code is the return value, that is to say the result of the function.

Suppose you want to create a very simple function that determines what percentage a given value `x` is of another value `y`. As you may have guessed, this function takes two arguments: the value whose percentage you want to determine (the numerator) and the value used as a reference point (the denominator). Let us call this function `percentage()`. This is how you write it:

```
> percentage <- function(x, y, ...){
+   result <- (x*100)/y
+   result
+ }
```

If you want to reuse the function, copy and paste it in a text file and save the file using the following name and extension: `function_name.r`. Right now, I advise you to store the function `percentage()` in a file named `mypercentage.r` in `CLSR/chap2`.

2.8.2.2 Loading the Function

Now that your function is defined, it is time to load it. You have two options: you can either copy and paste the code of your function into the R console or you can source your function from a file using `source()`. If you adopt the second option, you have two more options. The first option is to prompt R to open a window so as to select the file interactively. Note that there is a slight difference between Windows users and Mac users. Where Windows users write `choose.files()` to open an interactive window, Mac users must write `file.choose()`:

```
> source(file=choose.files()) # Windows
> source(file=file.choose()) # Mac
```

The second option is to enter the path of your function file as an argument of `source()` with quotes:

```
> source("C:/CLSR/function_name.r") # Windows
> source("/CLSR/function_name.r") # Mac
```

These two options are available whenever a function has a `file` argument.

2.8.2.3 Running the Function

Now that your function is loaded into R, you can run it. Suppose you want to know what percent 24 is of 256. Given how you assigned arguments in your function, 24 should be the first argument (`x`) and 256 should be the second argument (`y`). These arguments should follow this order in the function's bracket:

```
> percentage(24, 256)
[1] 9.375
```

2.8.2.4 Fine-Tuning the Function

Your function works fine, but now that it is stored safely on your computer, I suggest we fine-tune it so that its output looks nicer. First, let us reduce the number of decimals to two with `round()`. Minimally, this function takes two arguments. The first argument is the numeric value you wish to round, and the second argument is the number of decimal places. All we need to do is take the result, which is already saved in the named data structure `result`, place it in first-argument position in `round()`, and set the second argument to 2 to tell R we only want two decimals:

```
> rounded_res <-round(result, 2) # reduce the number of decimals to 2
```

Next, let us embed the result in some explanatory text, such as "`x` is X.XX percent of `y`". To do it, we can use the function `cat()`, which prints multiple objects, one after the other. As the argument `sep=" "` indicates, each object is separated by a space.

```
> cat(x, "is", rounded_res,"percent of", y, sep=" ")
```

Because `x`, `rounded_res`, and `y` are named data structures, `cat()` will print their respective values. On the other hand, `"is"` and `"percent of"` are character strings, as signaled by the quotes (see Sect. 2.9.1.1 below). The values and the text elements are separated by a space, as specified in the `sep` argument (i.e. there is a space between the quotes). Now, the function looks like this:

```
percentage <- function(x, y, ...){
       result <- (x*100)/y
       rounded_res <-round(result, 2)
       cat(x, "is", rounded_res,"percent of", y)
}
```

Let us run it again to see the changes:

```
> percentage(24,256)
24 is 9.38 percent of 256
```

The output is definitely more user friendly. This fine-tuned function is already stored in your `CLSR` folder.

User-defined functions are interesting to corpus linguists because they can save a lot of time when it comes to repeating identical operations. We will write functions on some occasions throughout the book.

2.9 R Objects

There are four main kinds of R objects: vectors, lists, matrices, and data frames. I will present each of them in turn. I will also introduce a fifth object (factors) when I discuss data frames.

2.9.1 Vectors

The vector is the most basic R object. It is an ordered sequence of data elements. There are three types of vectors: character vectors, numeric vectors, and logical vectors. Vectors also have a length.

2.9.1.1 Vector Modes

Vectors come in three modes: character, numeric, and logical vectors. Character vectors are strings of characters. They are delimited by single or double quotes. In this book, I use double quotes:

```
> char_vec <- "a character vector" ; char_vec
[1] "a character vector"
```

Outside linguistics and text analysis, character vectors are mostly used for labels (e.g. in plots). Linguists exploit character vectors more systematically in corpus compiling and corpus exploration. An interesting property of character vectors that contain alphabetic characters is that you can easily convert the strings between lower case and upper case. This is done with two functions: `toupper()` and `tolower()`.

```
> char_vec <- toupper(char_vec) ; char_vec # converting char_vec from lower case to upper case
[1] "A CHARACTER VECTOR"
> char_vec <- tolower(char_vec) ; char_vec # converting char_vec back to lower case
[1] "a character vector"
```

Obviously, numeric vectors contain numbers, without quotes.

```
> num_vec <- 10 ; num_vec
[1] 10
```

Logical vectors contain Boolean values, namely the strings `TRUE` and `FALSE`, without quotes.

```
> logi_vec <- FALSE ; logi_vec
[1] FALSE
```

When there are no quotes, R recognizes `FALSE` as a logical value. This value can be abbreviated to `F`:

```
> logi_vec <- F ; logi_vec
[1] FALSE
```

2.9.1.2 Vector Mode Conversion

One interesting feature regarding vectors (and parts of other R objects) is mode conversion thanks to three functions: `as.numeric()`, `as.character()`, and `as.logical()`. A character vector can be converted into a numeric vector if it contains elements that can be treated as numeric values. Below, three vectors are concatenated into the vector `v` by means of the function `c()` (see Sect. 2.9.1.3).

```
> v <- c("3", "2", "1") ; v
[1] "3" "2" "1"
```

You can verify that `v` has indeed been recognized as a character vector by displaying its structure with either `mode()`,

```
> mode(v)
[1] "character"
```

`class()`,

```
> class(v)
[1] "character"
```

or, better, `str()` (for "structure").

```
> str(v) # display the vector mode of v + its internal structure
 chr [1:3] "3" "2" "1"
```

You may also ask R if `v` is a character vector with `is.character()`:

```
> is.character(v)
[1] TRUE
```

As you may have guessed from its syntax, this question is actually a function. R answers `TRUE`, which is its way of saying "yes" (and `FALSE` means "no"). You may now proceed to the conversion and store it in R's memory by assigning `as.numeric(v)` to the vector `v`:

```
> v <- as.numeric(v) ; v
[1] 3 2 1
```

The functions `str(v)` and `is.numeric(v)` confirm that the conversion has been successful.

```
> str(v)
 num [1:3] 3 2 1
> is.numeric(v)
[1] TRUE
```

If the vector contains a mix of numeric and non-numeric values, the conversion will only apply to those elements that can be converted into numeric values. For the other elements of the vector, NA values will be generated (NA stands for "Not Applicable"), and R will let you know:

```
> v2 <- c("3","2","d", "f")
> v2 <- as.numeric(v2) ; v2
[1]  3  2 NA NA
```

A character vector can also be converted into a logical vector if it contains the following character strings: "TRUE", "FALSE", "T", "F", "true", "false", "NA", or "na". R can easily recognize these character strings and convert them into their Boolean equivalents.

```
> v3 <- c("TRUE", "FALSE", "T", "F", "true", "false", "NA", "na")
> v3 <- as.logical(v3) ; v3
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE    NA
[8]    NA
```

Since you want to check if the vector is logical, use the `is.logical()` function.

```
> is.logical(v3)
[1] TRUE
```

A numeric vector can be converted into a character vector:

```
> v4 <- c(3, 2, 1)
> str(v4)
 num [1:3] 3 2 1
> v4 <- as.character(v4)
> str(v4)
 chr [1:3] "3" "2" "1"
```

The numbers now appear between quotes, which means they have been transformed into characters. A numeric vector can also be transformed into a logical vector. In this case, 0 will be interpreted as FALSE and all the other numbers as TRUE:

```
> v5 <- c(0,1,2,3,10,100,1000)
> str(v5)
 num [1:7] 0 1 2 3 10 100 1000
> v5 <- as.logical(v5)
> str(v5)
 logi [1:7] FALSE TRUE TRUE TRUE TRUE TRUE ...
```

When a logical vector is converted into a character vector, the Boolean values TRUE and FALSE are transformed into character strings, as evidenced by the presence of quotes. NA remains unchanged because R thinks you want to signal a missing value:

```
> v6 <- c(TRUE, TRUE, T, F, FALSE, NA)
> str(v6)
 logi [1:6] TRUE TRUE TRUE FALSE FALSE NA
> v6 <- as.character(v6)
> str(v6)
 chr [1:6] "TRUE" "TRUE" "TRUE" ...
```

When a logical vector is converted into a numeric vector, TRUE becomes 1 and FALSE becomes 0. NA remains unchanged for the same reason as above:

```
> v7 <- c(TRUE, TRUE, T, F, FALSE, NA)
> str(v7)
 logi [1:6] TRUE TRUE TRUE FALSE FALSE NA
> v7 <- as.numeric(v7)
> str(v7)
 num [1:6] 1 1 1 0 0 NA
```

Before running a concatenation, the function `c()` coerces its arguments so that they belong to the same mode. When there is a competition between several modes, the character mode is always given the highest priority,

```
> v8 <- c(0, 1, "sixteen", TRUE, FALSE) # numeric, character, and logical values
> str(v8)
 chr [1:5] "0" "1" "sixteen" "TRUE" ...
```

and the numeric mode has priority over the logical mode.

```
> v9 <- c(0,1,16,TRUE, FALSE) # numeric and logical values
> str(v9)
 num [1:5] 0 1 16 1 0
```

2.9.1.3 Vector Lengths

Vectors have a length, which you can measure with the function `length()`. It takes as argument a vector of any mode and length and returns a numeric vector of length 1.[7] All the vectors seen in Sect. 2.9.1.1 are of length 1 because they consist of one element.

```
> length(char_vec)
[1] 1
> length(num_vec)
[1] 1
> length(logi_vec)
[1] 1
```

If you are new to R, you may find the result surprising for `char_vec` because the vector consists of three words separated by spaces. You may expect a length of 3. Yet, `length()` tells us that the vector consists of one element. This is because R does not see three words, but a single character string. Because R does not make an a priori distinction between spaces and word characters (or letters), we are going to have to tell R what we consider a word is at some point.[8]

R has a function to count the number of characters in a string: `nchar()`, which minimally takes the vector as an argument. If R treated only word characters as arguments, `nchar()` would do the count as follows...

a		c	h	a	r	a	c	t	e	r		v	e	c	t	o	r
1		2	3	4	5	6	7	8	9	10		11	12	13	14	15	16

...and `nchar(char_vec)` would return 16. Since `nchar()` treats spaces as characters, it counts characters as follows...

a		c	h	a	r	a	c	t	e	r		v	e	c	t	o	r
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

...and `nchar(char_vec)` returns 18.

[7] Its argument can also be any R object, under conditions. See `?length`.

[8] The question of what counts as a word is not trivial. Haspelmath (2011) argues that a word can only be "defined as a language-specific concept". Corpus linguists must therefore provide the computer with an ad hoc definition for each language that they work on.

```
> nchar(char_vec)
[1] 18
```

To sum up, `char_vec` is a vector of length 1 that consists of 18 characters. For `char_vec` to be one vector of length 3, it would have to be the combination of three vectors of length 1. Let us create these vectors:

```
> vector1 <- "a"
> vector2 <- "character"
> vector3 <- "vector"
```

To combine *n* vectors of length 1 to one vector of length *n*, use the function `c()` (for "combine"):

```
> vector4 <- c(vector1, vector2, vector3) ; vector4
[1] "a"         "character" "vector"
```

Here is an alternative:

```
> vector4 <- c("a","character","vector") ; vector4
[1] "a"         "character" "vector"
```

This time, the length of `vector4` is 3:

```
> length(vector4)
[1] 3
```

The `c()` function also works with numeric and logical vectors:

```
> vector5 <- c(1,2,3,4,5,10,100,1000) # a numeric vector of length 8
> length(vector5)
[1] 8
> vector6 <- c(TRUE, TRUE, FALSE, FALSE, NA) # a logical vector of length 5
> length(vector6)
[1] 5
```

The function `paste()` is specific to character vectors. Its first argument is one or more R objects to be converted to character vectors. Its second argument is `sep`, the character string used to separate the vector elements (the space is the default). To merge several character vectors of length 1 into a single character vector of length 1, `sep` must have its default setting, i.e. a space.

```
> paste("now", "is", "the", "winter", "of", "our", "discontent", sep=" ")
[1] "now is the winter of our discontent"
```

The third, optional, argument of `paste()` is `collapse`, the character string used to separate the elements when you want to convert a vector of length *n* into a vector of length 1. In this case, you must set `collapse` to a space because the default is `NULL`.

```
> richard3 <-  c("now", "is", "the", "winter", "of", "our", "discontent")
> paste(richard3, collapse=" ")
[1] "now is the winter of our discontent"
```

2.9.1.4 Manipulating Vectors

Creating a vector manually is simple, and you already know the drill:

- for a vector of length 1, enter a character/numeric/logical value and assign it to a named vector with `<-`;

- for a vector of length $n > 1$, concatenate character/numeric/logical values with `c()` and assign them to a named vector with `<-`.

Let me show you a few more functions that come in handy when you create or manipulate vectors.

The function `seq()` allows you to create a numeric vector that contains a regular sequence of numbers. Minimally, it takes two arguments. The first argument (`from`) is the starting point of the sequence. The second argument (`to`) is the endpoint of your sequence. By default, the increment is 1. For example, `seq(1,10)` generates a sequence of ten integers comprised between 1 and 10:

```
> seq(1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

When the increment is 1, there is a shortcut version of `seq()`:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

Optionally, you can change the increment by adding a third argument (`by`). Let us set it to 2:

```
> seq(1,10,2)
[1] 1 3 5 7 9
```

The function `rep()` replicates the value(s) in its first argument, which can be a vector. The second argument specifies how many times the first argument is replicated.

```
> rep("here we go", 3)
[1] "here we go" "here we go" "here we go"
```

Above, the character vector of length 1 `"here we go"` is replicated three times in a vector of length 3. Do you see what happens when a sequence is embedded into a replication?

```
> rep(1:3, 3)
[1] 1 2 3 1 2 3 1 2 3
```

The sequence 1 2 3 is replicated three times in a row.

Vectors can be sorted with `sort()`.[9] Suppose you have a vector containing the names of some major linguists.

```
> linguists <- c("Langacker", "Chomsky", "Lakoff", "Pinker", "Jackendoff", "Goldberg")
```

To sort the names in ascending alphabetical order, just place `linguists` in argument position.

```
> sort(linguists)
[1] "Chomsky"    "Goldberg"   "Jackendoff"
[4] "Lakoff"     "Langacker"  "Pinker"
```

If you want to sort the names in descending alphabetical order, add `decreasing=TRUE`.

```
> sort(linguists, decreasing=TRUE)
[1] "Pinker"     "Langacker"  "Lakoff"
[4] "Jackendoff" "Goldberg"   "Chomsky"
```

Suppose now that you have a vector with the years of birth of the participants in a psycholinguistic experiment.

[9] In linguistics, you generally sort character and numeric vectors.

```
> years <- c(1990, 1995, 1988, 1961, 1937, 1992, 1976, 1977)
```

Again, use `sort()` to sort the years in ascending or descending order.

```
> sort(years) # ascending order
[1] 1937 1961 1976 1977 1988 1990 1992 1995
> sort(years, decreasing = TRUE) # descending order
[1] 1995 1992 1990 1988 1977 1976 1961 1937
```

When a vector is of length $n > 1$, you can easily extract one or several elements from the vector. Extraction is made possible thanks to how R indexes the internal structure of vectors. It is done thanks to square brackets `[]`. Let us first create a character vector of length 6:

```
> v10 <- c("I", "bet", "you", "already", "like", "R")
```

Each element of the vector is indexed as follows:

I	bet	you	already	like	R
[1]	[2]	[3]	[4]	[5]	[6]

To extract the fourth element, which we expect to be `"already"`, all we need to do is specify the index:

```
> v10[4]
[1] "already"
```

To extract several parts of the vector, embed `c()` and make your selection:

```
> v10[c(1, 5, 6)]
[1] "I"    "like" "R"
> v10[c(3, 5, 6)]
[1] "you"  "like" "R"
```

You can also rearrange a selection of vector elements:

```
> v10[c(1, 5, 3)]
[1] "I"    "like" "you"
```

The same techniques also allows for the substitution of vector elements:

```
> v10[c(2,5)] <- c("know", "love"); v10
[1] "I"       "know"    "you"     "already"
[5] "love"    "R"
```

An interesting property of vector elements is that you can name them with the function `names()`. This function does not appear to the right of `<-` but to its left. Its argument is the vector whose elements you want to name.

```
> v11 <- c("thank you", "merci", "danke", "grazie", "gracias")
> names(v11) <- c("English", "French", "German", "Italian", "Spanish")
```

Now, each element of `v11` is named,

```
> v11
    English      French      German     Italian
"thank you"     "merci"     "danke"    "grazie"
    Spanish
  "gracias"
```

and you may use the names to extract vector elements.

```
> v11[c("French", "Spanish")]
   French   Spanish
  "merci" "gracias"
```

When two vectors of different lengths are involved in an operation, R recycles the values of the shorter vector until the length of the longer vector is matched. This is called recycling. Do you understand what happens below?

```
> c(1:5)*2
[1]  2  4  6  8 10
```

The shorter vector `2` is recycled to multiply each element of the sequence vector by 2. Here is another example where the length of the shorter vector is greater than 1. Again, do you understand what R does?

```
> c(1:5)*c(2,3)
[1]  2  6  6 12 10
```

R multiplies each element of the sequence by either 2 or 3, in alternation: 1×2, 2×3, 3×2, 4×3, and 5×2.

2.9.1.5 Logical Operators

Thanks to logical operators, you can tell R to decide which vector elements satisfy particular conditions.[10] Tab. 2.1 displays the most common logical operators.

Table 2.1: Logical operators in R

operator	what it means
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	exactly equal to
`!=`	not equal to
`!x`	Not x
`x | y`	x OR y
`x & y`	x AND y

Often, logical operators are used as arguments of the function `which()`, which outputs the positions of the elements in the vector that satisfy a given condition.

```
> v12<-seq(0,20,2) ; v12 # a sequence from 0 to 20 with increment 2
 [1]  0  2  4  6  8 10 12 14 16 18 20
> which(v12>=6) # which element is greater than or equal to 6?
[1]  4  5  6  7  8  9 10 11
```

[10] Logical operators are not specific to R. They are found in most programming languages.

Bear in mind that `which` outputs the *positions* of the elements that satisfy the condition "greater than or equal to 6", not the elements themselves.

```
> which(v12 < 4 | v12 > 14) # which elements of v12 are less than 4 OR greater than 14?
[1]  1  2  9 10 11
> which(v12 > 2 & v12 < 10) # which elements of v12 are greater than 2 AND less than 10?
[1] 3 4 5
> which(v12!=0) # which elements of v12 are not 0?
 [1]  2  3  4  5  6  7  8  9 10 11
```

Logical operators also work with character vectors.

```
> which(v11!="grazie") # which elements of v11 are not "grazie"
English  French  German Spanish
      1       2       3       5
> which(v11=="danke"|v11=="grazie") # which elements of v11 are "danke" or "grazie"
 German Italian
      3       4
```

Each time, the names are preserved. They appear on top of the relevant vector positions.

2.9.1.6 Loading Vectors

If the vector is saved in an existing file, you can load it using `scan()`. In the example below, we assume the vector to be loaded is a character vector.

```
> load_v <- scan(file=choose.files(), what="char", sep="\n") # Windows
> load_v <- scan(file=file.choose(), what="char", sep="\n") # Mac
```

As you already know, `choose.files()` and `file.choose()` open a window to select the file interactively. The argument `file` can also be a path. The code below opens an example character vector stored in the text file named `example_character_vector.txt` in your `CLSR` folder:

```
> load_v <- scan(file="C:/CLSR/chap2/example_character_vector.txt", what="char", sep="\n") # Windows
> load_v <- scan(file="/CLSR/chap2/example_character_vector.txt", what="char", sep="\n") # Mac
```

The argument `what` is set to `"char"`, which stands for "character strings". Finally, `sep` is set to `"\n"` for "new line". This means that R uses line breaks to delimit vector elements. In other words, `scan()` expects to read new-line delimited vector elements. Given this specification, the vector `load_v` has three elements (= `load_v` is of length 3):

```
> length(load_v)
[1] 3
```

Other common separators are: `" "` (a space) and `"\t"` (a tab stop). If you set `sep` to a space `" "`...

```
> load_v2 <- scan(file="C:/CLSR/chap2/example_character_vector.txt", what="char", sep=" ") # Windows
> load_v2 <- scan(file="/CLSR/chap2/example_character_vector.txt", what="char", sep=" ") # Mac
```

... the length of the vector changes accordingly.

```
> length(load_v2)
[1] 19
```

2.9.1.7 Saving Vectors

To save a vector, use the `cat()` function seen above (Sect. 2.8.2.4).

```
> cat(... , file, sep = " ", append = FALSE)
```

- `...`: an R object (here the vector you want to output);
- `file`: the file to print to (if you do not provide it, the vector is printed onto the R console);
- `sep`: the separator to append after each element (because generally you want each vector to have its own line, it is best to set `sep` to `"\n"`);
- `append`: whether you want to append the output to pre-existing data in a file.

Let us save `v11` in a plain text file named `my.saved.vector.txt` in your `CLSR` folder. The file does not exist yet, but R will create it for you because you have provided the `.txt` extension.

```
> cat(v11, file="C:/CLSR/chap2/my.saved.vector.txt", sep="\n") # Windows
> cat(v11, file="/CLSR/chap2/my.saved.vector.txt", sep="\n") # Mac
```

Open your `CLSR` folder to check if the vector has been saved properly.

2.9.2 Lists

Many R functions used in corpus linguistics and statistics return values that are lists. A list is a data structure that can contain R objects of varying modes and lengths. Suppose you have collected information regarding six corpora.[11] For each corpus, you have the name, the size in million words, and some dialectal information as to whether the texts are in American English. You also have the total size of all the corpora. Each item is stored in a separate vector.

```
> corpora <- c("COCA", "COHA", "TIME", "American Soap", "BNC", "Strathy")
> size <- c(450, 400, 100, 100, 100, 50)
> us_english <- c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE)
> total_size <- 1200
```

All these vectors can be stored in a list using the `list()` function.

```
> yourlist <- list(corpora, size, us_english, total_size)
```

List components can be named as you create the list.

```
> yourlist <- list(corpora=corpora, size_in_M_words=size, is_dialect_us=us_english,
+                  total_size=total_size)
> yourlist
$corpora
[1] "COCA"          "COHA"
[3] "TIME"          "American Soap"
[5] "BNC"           "Strathy"

$size_in_M_words
[1] 450 400 100 100 100  50

$is_dialect_us
```

[11] All of them can be accessed here: http://corpus.byu.edu/.

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

$total_size
[1] 1200
```

Components can be accessed by name using `$`, or by position using double square brackets `[[]]`.

```
> yourlist$size # the second element accessed by name
[1] 450 400 100 100 100  50
> yourlist[[2]] # the second element accessed by position
[1] 450 400 100 100 100  50
```

Sometimes, you will want to extract individual components from the list so as to manipulate them. This is done with `unlist()`.

```
> unlist(yourlist[[2]])
[1] 450 400 100 100 100  50
```

The output is a vector containing the values of the second list component.

2.9.3 Matrices

A matrix is a two-dimensional table. Tab. 2.2 is a good example of what can be loaded into R in the form of a matrix. Its entries are the frequency counts of three patterns (<*each* + N>, <*every* + N>, and <*each and every* + N>) in three corpora of English: the British National Corpus, the Corpus of Contemporary American English, and the Corpus of Web-Based Global English.[12]

Table 2.2: An example matrix (<*each* + N>, <*every* + N>, and <*each and every* + N> in three corpora of English)

	BNC	COCA	GloWbE
each + N	30708	141012	535316
every + N	28481	181127	857726
each and every + N	140	907	14529

To create a matrix, use the `matrix()` function.

```
> matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
+        dimnames = NULL)
```

- `x`: a vector of values
- `nrow`: the number of rows
- `ncol`: the number of columns
- `byrow`: how you enter the values in the matrix
- `dimnames`: a list containing vectors of names for rows and columns

[12] http://corpus.byu.edu/.

To make a matrix from Tab. 2.2, start with the vector of values. By default, the matrix is filled by columns (from the leftmost to the rightmost column, and from the top row to the bottom row).

```
> values <- c(30708, 28481, 140, 141012, 181127, 907, 535316, 857726, 14529)
```

Embed the vector in the `matrix()` function.

```
> mat <- matrix(values, 3, 3); mat
      [,1]   [,2]   [,3]
[1,] 30708 141012 535316
[2,] 28481 181127 857726
[3,]   140    907  14529
```

The data is indexed in a [row, column] fashion. To access the value in the second row of the third column (i.e. 857726), enter:

```
> mat[2,3]
[1] 857726
```

This allows you to perform calculations. For example, if you want to know what percentage the frequency of *each + N* in the BNC is of the frequency of the same pattern in the GloWbE, you may reuse the `percentage()` function from Sect. 2.8.2.

```
> percentage(mat[1,1], mat[1,3])
30708 is 5.74 percent of 535316
```

You may calculate row sums and column sums with `rowSums()` and `colSums()` respectively.

```
> rowSums(mat)
[1]  707036 1067334   15576
> colSums(mat)
[1]   59329  323046 1407571
```

You can sum the whole matrix with `sum()`

```
> sum(mat)
[1] 1789946
```

The result is the sum of all row totals or column totals.

Optionally, you can add the row names and the column names with the argument `dimnames`. These names should be stored in two vectors beforehand.

```
> row_names <- c("each + N", "every + N", "each and every + N")
> col_names <- c("BNC", "COCA", "GloWbE")
```

Bear in mind that `dimnames` must be a list of length 2.[13] The vectors `row_names` and `col_names` must therefore be placed in a list.

```
> mat<-matrix(values, 3, 3, dimnames=list(row_names, col_names))
> mat
                     BNC   COCA GloWbE
each + N           30708 141012 535316
every + N          28481 181127 857726
each and every + N   140    907  14529
```

[13] If `dimnames` is of length 1, it is assumed that the list contains only row names.

2.9.4 Data Frames (and Factors)

In quantitative corpus linguistics, analyses involve a set of related observations which are grouped into a single object called a data set. For example, you might collect information about two linguistic units so as to compare them in the light of descriptive variables (qualitative and/or quantitative).

2.9.4.1 Ready-Made Data Frames in Plain Text Format

For illustrative purposes, let me anticipate on a case study that we will come back to: *quite* and *rather* in the British National Corpus. Typically, you extract observations containing the units and you annotate each observation for a number of variables. With regard to *quite* and *rather*, these variables can be:

- the name of the corpus file and some information about it,
- the intensifier in context (*quite* or *rather*),
- the intensifier out of context,
- the syntax of the intensifier (preadjectival or pre-determiner),
- the intensified adjective co-occurring with the intensifier and the NP modified by the adjective,
- the syllable count of the adjective and the NP.

The above translates into a data set, sampled in Tab. 2.3. If you store the data set in a matrix, with observations in the rows and variables in the columns, you will benefit from the rigorous indexing system to access the data, but the matrix will find it difficult to accommodate different modes. It will for instance coerce numeric variables such as `syllable count ADJ` and `syllable count NP` into character variables, which we do not necessarily want (don't forget, you decide and R executes). This is where the data frame steps in: it combines the ease of indexing of a matrix with the accommodation of different modes, providing each column displays variables of a single mode.

Table 2.3: A sample data frame (*quite* and *rather* in the BNC)

corpus_file	corpus_file_info	exact match	intensifier	syntax	adjective	syllable_count_adj	NP	syllable_count_NP
KBF.xml	S conv	a quite ferocious mess	quite	preadjectival	ferocious	3	mess	1
AT1.xml	W biography	quite a flirty person	quite	predeterminer	flirty	2	person	2
A7F.xml	W misc	a rather anonymous name	rather	preadjectival	anonymous	4	name	1
ECD.xml	W commerce	a rather precarious foothold	rather	preadjectival	precarious	4	foothold	2
B2E.xml	W biography	quite a restless night	quite	predeterminer	restless	2	night	1
AM4.xml	W misc	a rather different turn	rather	preadjectival	different	3	turn	1
F85.xml	S unclassified	a rather younger age	rather	preadjectival	younger	2	age	1
J3X.xml	S unclassified	quite a long time	quite	predeterminer	long	1	time	1
KBK.xml	S conv	quite a leading light	quite	predeterminer	leading	2	light	1
EC8.xml	W nonAc: humanities arts	a rather different effect	rather	preadjectival	different	3	effect	2

The sample data set is available in your `CLSR` folder in a file named `sample.df.txt`. As its extension indicates, it is a text file.[14] Open it with your regular text editor and make sure that it shows invisibles such as spaces and tab stops (for example, in TextWrangler: `View > Text Display > Show Invisibles`).

[14] Beside `.txt`, `.csv` (for "comma separated file") is a common extension for data frames stored in plain text format.

You should see something like Fig. 2.1, where triangles represent tab stops and the symbol "¬" new lines. Each column is separated by a tab stop. The text file is hardly legible as such, but if you import the file into a spreadsheet software, each element delimited with a tab stop will be assigned its own column. As you can see in Fig. 2.2, the spreadsheet display is more user friendly.

```
corpus_file▵corpus_file_info▵   match▵  intensifier▵construction▵   adjective▵  syllable_count_adj▵ NP▵ syllable_count_NP¬
KBF.xml▵S·conv▵ a·quite·ferocious·mess·▵quite▵  preadjectival▵  ferocious▵  3▵  mess▵   1¬
AT1.xml▵W·biography▵quite·a·flirty·person▵  quite▵  predeterminer▵  flirty▵ 2▵  person▵ 2¬
A7F.xml▵W·misc▵ a·rather·anonymous·name·▵   rather▵ preadjectival▵  anonymous▵  4▵  name▵   1¬
ECD.xml▵W·commerce▵ a·rather·precarious·foothold▵   rather▵ preadjectival▵  precarious▵ 4▵  foothold▵   2¬
B2E.xml▵W·biography▵quite·a·restless·night▵ quite▵  predeterminer▵  restless▵   2▵  night▵  1¬
AM4.xml▵W·misc▵ a·rather·different·turn▵rather▵ preadjectival▵  different▵  3▵  turn▵   1¬
F85.xml▵S·unclassified▵ a·rather·younger·age·▵  rather▵ preadjectival▵  younger▵2▵  age▵1¬
J3X.xml▵S·unclassified▵ quite·a·long·time·▵ quite▵  predeterminer▵  long▵   1▵  time▵   1¬
KBK.xml▵S·conv▵ quite·a·leading·light▵  quite▵  predeterminer▵  leading▵2▵  light▵  1¬
EC8.xml▵W·nonAc:·humanities·arts▵   a·rather·different·effect·▵ rather▵ preadjectival▵  different▵  3▵  effect▵ 2¬
```

Fig. 2.1: The sample data frame stored in plain text format when tab stops and new lines are made visible

	A	B	C	D	E	F	G	H	I
1	corpus_file	corpus_file_info	match	intensifier	construction	adjective	syllable_count_adj	NP	syllable_count_NP
2	KBF.xml	S conv	a quite ferocious mess	quite	preadjectival	ferocious	3	mess	1
3	AT1.xml	W biography	quite a flirty person	quite	predeterminer	flirty	2	person	2
4	A7F.xml	W misc	a rather anonymous name	rather	preadjectival	anonymous	4	name	1
5	ECD.xml	W commerce	a rather precarious foothold	rather	preadjectival	precarious	4	foothold	2
6	B2E.xml	W biography	quite a restless night	quite	predeterminer	restless	2	night	1
7	AM4.xml	W misc	a rather different turn	rather	preadjectival	different	3	turn	1
8	F85.xml	S unclassified	a rather younger age	rather	preadjectival	younger	2	age	1
9	J3X.xml	S unclassified	quite a long time	quite	predeterminer	long	1	time	1
10	KBK.xml	S conv	quite a leading light	quite	predeterminer	leading	2	light	1
11	EC8.xml	W nonAc: humanities arts	a rather different effect	rather	preadjectival	different	3	effect	2

Fig. 2.2: The sample data frame displayed in a spreadsheet software

When you export corpus observations into a data frame for later treatment in R or in a spreadsheet software, or when you load a ready-made data frame into R, you must specify a delimiter, as you will see below. Beside tab stops, common delimiters are: the comma, the semicolon, the space, and the whitespace.

To load a data frame, you may use `read.table()`. Because it has a large array of argument options, it is very flexible (type `?read.table` to see them), but for the same reason, it is sometimes tricky to use. The essential arguments of `read.table()` are the following:

```
> read.table(file, header = TRUE, sep = "\t", row.names=NULL)
```

- `file`: the path of the file from which the data set is to be read;
- `header=TRUE`: your dataset contains column headers in the first row; R sets it to `TRUE` automatically if the first row contains one fewer field than the number of columns;
- `sep="\t"`: the field delimiter (here, a tab stop);
- `row.names=NULL`: by default, R numbers each row.

The tricky part concerns the compulsory attribution of row names. If you specify `row.names`, you must provide the row names yourself. They can be in a character vector whose length corresponds to the number of rows in your data set. The specification can also be the number of the column that contains the row names. For example, `row.names=1` tells R that the row names of the data frame are in the first column. When

you do this, you must be aware that R does not accept duplicate row names or missing values. Imagine you collect 1000 utterances from 4 speakers, each speaker contributing 250 utterances. You cannot use the speakers' names as row names because each will be repeated 250 times, and the data frame will not load. Because `sample.df.txt` does not have a column that contains row names, we stick to the default setting of `row.names`. To load it, we just enter the following:

```
> df <- read.table("C:/CLSRchap2/sample.df.txt", header=TRUE, sep="\t") # Windows
> df <- read.table("/CLSR/chap2/sample.df.txt", header=TRUE, sep="\t") # Mac
```

In my experience, specialized implementations of `read.table()`, such as `read.csv()` or `read.delim()`, are easier to use. My favorite is `read.delim()`.

```
> read.delim(file, header = TRUE, sep = "\t", quote = "\"",
+            dec = ".", fill = TRUE, comment.char = "", ...)
```

When your text file is formatted as in Fig. 2.1, loading a data frame always works with `read.delim()` and its default argument settings.

```
> df <- read.delim("C:/CLSR/chap2/sample.df.txt") # Windows
> df <- read.delim("/CLSR/chap2/sample.df.txt") # Mac
```

You may now inspect the structure of the data frame by entering `str(df)`.

```
> str(df)
'data.frame': 10 obs. of  9 variables:
 $ corpus_file       : Factor w/ 10 levels "A7F.xml","AM4.xml",..: 9 3 1 6 4 2 7 8 10 5
 $ corpus_file_info  : Factor w/ 6 levels "S conv","S unclassified",..: 1 3 5 4 3 5 2 2 1 6
 $ match             : Factor w/ 10 levels "a quite ferocious mess ",..: 1 7 2 5 10 4 6 9 8 3
 $ intensifier       : Factor w/ 2 levels "quite","rather": 1 1 2 2 1 2 2 1 1 2
 $ construction      : Factor w/ 2 levels "preadjectival",..: 1 2 1 1 2 1 1 2 2 1
 $ adjective         : Factor w/ 9 levels "anonymous","different",..: 3 4 1 7 8 2 9 6 5 2
 $ syllable_count_adj: int  3 2 4 4 2 3 2 1 2 3
 $ NP                : Factor w/ 10 levels "age","effect",..: 5 8 6 3 7 10 1 9 4 2
 $ syllable_count_NP : int  1 2 1 2 1 1 1 1 1 2
```

The function outputs:

- the number of observations (rows),
- the number of variables (columns),
- the type of each variable.

The data set contains two integer type variables[15] and seven factor type variables. A factor variable contains either nominal or ordinal values (the factors in this data set has only nominal variables). Nominal values are unordered categories whereas ordinal variables are ordered categories with no quantification between them.[16] By default, R converts all nominal and ordinal variables into factors. Factors have levels, i.e. unique values. For example, the factor `corpus_file_info` has six unique values, three of which appear twice (`S conv`, `W biography`, and `S unclassified`).

Because a data frame is indexed in a [row, column] fashion (like a matrix), extracting data points is easy as pie. For instance, to extract the adjective *restless*, which is in the fifth row of the sixth column, enter:

[15] Integer is a subtype of numeric.

[16] If you grade people's language proficiency on a scale from 0 to 5, you have an ordinal variable containing ordered values (5 is higher than 4, which is higher than 3, etc.), but the difference between these values is not proportional. For example, someone whose proficiency level is graded as 3 is not necessarily three times as proficient as someone whose proficiency level is graded as 1.

```
> df[5,6]
[1] restless
9 Levels: anonymous different ... younger
```

or use the variable name between quotes.

```
> df[5,"adjective"]
[1] restless
9 Levels: anonymous different ... younger
```

To extract a variable from the data frame, use the `$` symbol.[17]

```
> df$adjective
 [1] ferocious  flirty     anonymous  precarious
 [5] restless   different  younger    long
 [9] leading    different
9 Levels: anonymous different ... younger
```

To extract the adjective *restless*, just add the row number between square brackets.

```
> df$adjective[5]
[1] restless
9 Levels: anonymous different ... younger
```

2.9.4.2 Generating a Data Frame Manually

When your data frame is small, you can enter the data manually. Suppose you want to enter Tab. 2.4 into R. Four corpora of English are described by three variables:

- the size in million words,
- the variety of English,
- the period covered by the corpus.

Table 2.4: Another example data frame

corpus	size	variety	period
BNC	100	GB	1980s-1993
COCA	450	US	1990-2012
Hansard	1600	GB	1803-2005
Strathy	50	CA	1970s-2000s

First, generate four vectors, one for each column of the data frame.

```
> corpus <- c("BNC", "COCA", "Hansard", "Strathy")
> size <- c(100, 450, 1600, 50)
> variety <- c("GB", "US", "GB", "CA")
> period <- c("1980s-1993", "1990-2012", "1803-2005", "1970s-2000s")
```

[17] As seen in Sect. 2.9.2, the dollar symbol `$` is also used to access list elements. This is because a data frame is similar to a list (enter `mode(df)` to see that the data frame is recognized as a list with regards to its mode).

You may now combine these four vectors into a data frame with the `data.frame()` function. If you want to use the values in the first column as row names, add the argument `row.names` to the function.

```
> df.manual <- data.frame(size, variety, period, row.names = corpus)
> df.manual
        size variety      period
BNC      100      GB  1980s-1993
COCA     450      US   1990-2012
Hansard 1600      GB   1803-2005
Strathy   50      CA 1970s-2000s
```

If you want the values in the first column to be treated as data points, not as row names, do not provide `row.names`.

```
> df.manual.2 <- data.frame(corpus, size, variety, period)
> df.manual.2 # R numbers the rows
   corpus size variety      period
1     BNC  100      GB  1980s-1993
2    COCA  450      US   1990-2012
3 Hansard 1600      GB   1803-2005
4 Strathy   50      CA 1970s-2000s
```

To export your data frame, use the `write.table()` function.[18] Just like `read.table()`, `write.table()` is elaborate and flexible (enter `?write.table` to see all the arguments that the function can take). For the task at hand, the following arguments will do:

```
> write.table(... , file, quote=FALSE, sep="\t", row.names=F)
```

- `...`: the R object to be exported as a data frame;
- `file`: the file to print to (if you do not provide it, the data frame is printed onto the R console);
- `quote`: whether you want character or factor columns to be surrounded by double quotes (`TRUE`) or not (`FALSE`);
- `sep`: the separator to append after each row element (here, a tab stop);
- `row.names`: if your first column contains data points (not row names), set `row.names` to `FALSE`.

To export `df.manual`, provide `col.names=NA` for a proper column alignment because `corpus` is used for row names.

```
> write.table(df.manual, file="C:/CLSR/chap2/my.saved.df.txt", quote=FALSE,
+             sep="\t", col.names=NA) # Windows
> write.table(df.manual, file="/CLSR/chap2/my.saved.df.txt", quote=FALSE,
+             sep="\t", col.names=NA) # Mac
```

To export `df.manual.2`, you have two options. If you want to preserve row numbering, provide `col.names=F`. Row numbers have a separate column.

```
> write.table(df.manual.2, file="C:/CLSR/chap2/my.saved.df.2a.txt", quote=FALSE,
+             sep="\t", col.names=NA) # Windows
> write.table(df.manual.2, file="/CLSR/chap2/my.saved.df.2a.txt", quote=FALSE,
+             sep="\t", col.names=NA) # Mac
```

If you do not want row numbering, provide `row.names=FALSE`.

```
> write.table(df.manual.2, file="C:/CLSR/chap2/my.saved.df.2b.txt", quote=FALSE,
+             sep="\t", row.names=FALSE) # Windows
> write.table(df.manual.2, file="/CLSR/chap2/my.saved.df.2b.txt", quote=FALSE,
+             sep="\t", row.names=FALSE) # Mac
```

[18] Although `write.csv()` exists, there is no `write.delim()` equivalent.

Your data frames are now saved in tab-delimited text format in your CLSR folder. Open my.saved.df.txt, my.saved.df.2a.txt, and my.saved.df.2b.txt with your spreadsheet software for inspection.

2.9.4.3 Loading and Saving a Data Frame as an R Data File

If you do not need to process or edit your data frame with a spreadsheet software, it is faster to load and save the data frame as an R data file with the .rds extension. To save df.manual.2 as an R data file use saveRDS(). The first argument is the named data structure containing the data frame. The second argument is the path to the file to save the data frame to.

```
> saveRDS(df.manual.2, file="C:/CLSR/chap2/df.manual.2.rds") # Windows
> saveRDS(df.manual.2, file="/CLSR/chap2/df.manual.2.rds") # Mac
```

The R data file is now saved in your CLSR folder. To load the file, use the readRDS() function. Its argument is the path of the file from which to read the data frame.

```
> readRDS("C:/CLSR/chap2/df.manual.2.rds") # Windows
> readRDS("/CLSR/chap2/df.manual.2.rds") # Mac
```

```
   corpus size variety      period
1     BNC  100      GB  1980s-1993
2    COCA  450      US   1990-2012
3 Hansard 1600      GB   1803-2005
4 Strathy   50      CA 1970s-2000s
```

2.9.4.4 Converting an R Object into a Data Frame

Some R objects can easily be converted into a data frame thanks to the as.data.frame() function. The only requirement is that the R object be compatible with the structure of a data frame. Let us convert the matrix mat into a data frame.

```
> mat.as.df <- as.data.frame(mat)
> str(mat.as.df)
'data.frame': 3 obs. of  3 variables:
 $ BNC   : num  30708 28481 140
 $ COCA  : num  141012 181127 907
 $ GloWbE: num  535316 857726 14529
```

2.10 for Loops

A loop iterates over an object *n* times to execute instructions. It is useful when the R object contains a very large number of elements. The general structure of a for loop depends on whether you have one instruction (one line of code) or more than one. With one instruction, the structure is the following.

```
> for (i in sequence) instruction
```

When there are several instructions, you need curly brackets to delimit the code over which the loop will work.

```
> #for (i in sequence) {
> #  instruction 1 # first line of code
> #  instruction 2 # second line of code
> #  ... # etc.
> #}
```

The loop consists of the keyword `for` followed by a pair of brackets. These brackets contain an identifier, here `i` (any name is fine). The identifier is followed by `in` and a vector of values to loop over. The vector of values is a sequence whose length is the number of times you want to repeat the instructions. The identifier `i` takes on each and every value in this vector. The instructions are between curly braces. The closing brace marks the exit from the loop.

Because the above paragraph is quite a handful, let us use a minimal example. Take the set of the first five letters of the alphabet.

```
> letters[1:5]
[1] "a" "b" "c" "d" "e"
```

The `for` loop in the code below

```
> for (i in letters[1:5]) print(i)
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
```

is equivalent to the following:

```
> i <- "a"
> print(i)
[1] "a"
> i <- "b"
> print(i)
[1] "b"
> i <- "c"
> print(i)
[1] "c"
> i <- "d"
> print(i)
[1] "d"
> i <- "e"
> print(i)
[1] "e"
```

Obviously, the `for` loop is more economical as `i` takes on the value of “a”, “b”, “c”, “d”, and “e” successively. With an additional line of code, the curly brackets are required.

```
> for (i in letters[1:5]) {
+   i <- toupper(i)
+   print(i)
+ }
[1] "A"
[1] "B"
[1] "C"
[1] "D"
[1] "E"
```

Loops are commonly used to cycle over rows of matrices or data frames (especially if there are many of rows). They also come in handy when you need to process a high number of files. The identifier is used to

take on each and every value in a vector that contains the names of the corpus files. As with almost anything in R, for loops can be nested.

One known issue with loops is that they get slower as the number of iterations increases. If the object to loop over is very large, make sure that you keep as many instructions outside the loop as you can. Because of this, loops have a bad reputation among R users. Indeed, you will sometimes have to look for alternatives to for loops: e.g. the iterators package, or the functions apply(), lapply(), tapply(), or sapply().

There are two other kinds of loops: while loops and repeat loops. To know more about them, enter ?Control in R.

2.11 if and if...else Statements

If you want to express a condition in R, use if statements. There are two kinds of if statements: simple if statements and if...else statements. Of course, if statements can be nested.

2.11.1 if Statements

The general structure of an if statement depends on the number of statements. If there is a unique statement, the structure is the following.

```
> if (condition) statement
```

If there are several statements, you need to use curly brackets.

```
> # if (condition) {
> #  statement 1 # first statement
> #  statement 2 # second statement
> #  ...
> #}
```

The vector integer contains negative and positive integers.

```
> integer <- c(-7, -4, -1, 0, 3, 12, 14)
```

Suppose you want to tag each negative integer in the vector with the character string "-> negative". Using a for loop, this is how you can do it.

```
> for (i in 1:length(integer)) {
+   if (integer[i] < 0) print(paste(integer[i], "-> negative"))
+ }
[1] "-7 -> negative"
[1] "-4 -> negative"
[1] "-1 -> negative"
```

The instructions in the loop are repeated as many times as there are elements in the vector integer, that is to say length(integer) times. The identifier i takes on each and every value in this vector from 1 to 7 (7 being the length of the vector). The if statement is placed between the curly braces of the for loop. The condition is delimited with brackets after if. It can be paraphrased as follows: "if the integer is strictly less than 0…". The statement that follows can be paraphrased as "print the integer, followed by the character string "-> negative"". The tag is appended only if the condition is true.

2.11.2 `if...else` Statements

When you want R to do one thing if a condition is true and another thing is the condition is false—rather than do nothing, as above—use an `if...else` statement. Suppose you want to tag those elements of the vector `integer` that are not negative with the character string "-> positive". All you need to do is add a statement preceded by `else` to tell R what to do if the condition is false.

```
> for (i in 1:length(integer)) {
+   if (integer[i] < 0) print(paste(integer[i], "-> negative"))
+   else print(paste(integer[i], "-> positive"))
+ }
[1] "-7 -> negative"
[1] "-4 -> negative"
[1] "-1 -> negative"
[1] "0 -> positive"
[1] "3 -> positive"
[1] "12 -> positive"
[1] "14 -> positive"
```

A similar result is obtained with `ifelse()`. The structure of this function is more compressed than an `if...else` statement. It is also a more efficient alternative to `for` loops.

```
> # ifelse(condition, what to do if the condition is true, what to do if the condition is false)
```

```
> ifelse(integer < 0, "-> negative", "-> positive")
[1] "-> negative" "-> negative" "-> negative"
[4] "-> positive" "-> positive" "-> positive"
[7] "-> positive"
```

The main difference between the `for` loop and the `ifelse()` function is the output. The former outputs as many vectors as there are iterations. The latter outputs one vector whose length corresponds to the number of iterations.

Because zero is neither negative nor positive, you should add a specific condition to make sure that 0 gets its own tag, e.g. "-> zero". That second `if` statement will have to be nested in the first `if` statement and appear after `else`.

```
> for (i in 1:length(integer)) {
+   if (integer[i] < 0) print(paste(integer[i], "-> negative")) # first if statement
+   else if (integer[i] == 0) print(paste(integer[i], "-> zero")) # second (nested) if statement
+   else print(paste(integer[i], "-> positive"))
+ }
[1] "-7 -> negative"
[1] "-4 -> negative"
[1] "-1 -> negative"
[1] "0 -> zero"
[1] "3 -> positive"
[1] "12 -> positive"
[1] "14 -> positive"
```

Once nested, `ifelse()` allows you to do the same operation (although with a different vector output) in one line of code.

```
> ifelse(integer < 0, "-> negative", ifelse(integer == 0, "-> zero", "-> positive"))
[1] "-> negative" "-> negative" "-> negative"
[4] "-> zero"     "-> positive" "-> positive"
[7] "-> positive"
```

The nested `ifelse` statement is in fact what R does if the condition of the first `ifelse` statement is false.

2.12 Cleanup

Right now, R is full of data structures, as you can see by typing `ls()`:

```
> ls()
 [1] "a"           "age"         "b"
 [4] "char_vec"    "col_names"   "corpora"
 [7] "corpus"      "df"          "df.manual"
[10] "df.manual.2" "i"           "integer"
[13] "linguists"   "load_v"      "load_v2"
[16] "logi_vec"    "mat"         "mat.as.df"
[19] "num_vec"     "percentage"  "period"
[22] "richard3"    "row_names"   "size"
[25] "total_size"  "us_english"  "v"
[28] "v10"         "v11"         "v12"
[31] "v2"          "v3"          "v4"
[34] "v5"          "v6"          "v7"
[37] "v8"          "v9"          "values"
[40] "variety"     "vector1"     "vector2"
[43] "vector3"     "vector4"     "vector5"
[46] "vector6"     "years"       "yourlist"
```

If you want to get rid of only one data structure, just type `rm()` or `remove()` and include the name of the structure you want to clear between the brackets:

```
> rm(a) # remove a
> ls() # a has been removed from the list
 [1] "age"          "b"            "char_vec"
 [4] "col_names"    "corpora"      "corpus"
 [7] "df"           "df.manual"    "df.manual.2"
[10] "i"            "integer"      "linguists"
[13] "load_v"       "load_v2"      "logi_vec"
[16] "mat"          "mat.as.df"    "num_vec"
[19] "percentage"   "period"       "richard3"
[22] "row_names"    "size"         "total_size"
[25] "us_english"   "v"            "v10"
[28] "v11"          "v12"          "v2"
[31] "v3"           "v4"           "v5"
[34] "v6"           "v7"           "v8"
[37] "v9"           "values"       "variety"
[40] "vector1"      "vector2"      "vector3"
[43] "vector4"      "vector5"      "vector6"
[46] "years"        "yourlist"
```

If you need to remove several data structures, separate the names of the data structures with a comma:

```
> rm(char_vec, num_vec, logi_vec) # remove char_vec, num_vec, and logi_vec
> ls() # char_vec, num_vec, and logi_vec have been removed from the list
 [1] "age"          "b"            "col_names"
 [4] "corpora"      "corpus"       "df"
 [7] "df.manual"    "df.manual.2"  "i"
[10] "integer"      "linguists"    "load_v"
[13] "load_v2"      "mat"          "mat.as.df"
[16] "percentage"   "period"       "richard3"
[19] "row_names"    "size"         "total_size"
[22] "us_english"   "v"            "v10"
[25] "v11"          "v12"          "v2"
[28] "v3"           "v4"           "v5"
[31] "v6"           "v7"           "v8"
[34] "v9"           "values"       "variety"
[37] "vector1"      "vector2"      "vector3"
[40] "vector4"      "vector5"      "vector6"
[43] "years"        "yourlist"
```

Before each new task, it is recommended that you clean up R's memory by entering:

```
> rm(list=ls(all=TRUE)) # cleans up R's memory
```

It is common practice to include the above line at the beginning of all your saved scripts. R's memory is now empty:

```
> ls()
character(0)
```

2.13 Common Mistakes and How to Avoid Them

Coding mistakes are common. Even experienced R users make them. Because R's error messages are rather cryptic, you need to review your code with a trained eye. Often, their resolution is straightforward. Here is a tentative checklist to help you out:

- R is unforgivingly case sensitive: the interpreter understands three different things when you type `mean()`, `Mean()`, and `MEAN()`. If you try the latter two, you are in for a cryptic error message.

  ```
  > Mean(c(2,5,10)) # wrong
  > MEAN(c(2,5,10)) # wrong
  ```

  Only the first option works.

  ```
  > mean(c(2,5,10))
  ```

- In the same vein, omitting compulsory quotes makes R unhappy.

  ```
  > install.packages(Hmisc) # it should be install.packages("Hmisc")
  ```

  R is more lenient regarding spaces.

  ```
  > mean(c(   2,  5,10   ))
  [1] 5.666667
  > mean (c(2,5,10))
  [1] 5.666667
  ```

- If you omit brackets when you call a function, R will print the function's code.

  ```
  > install.packages # hang in there!
  ```

  If you omit something that R can predict (e.g. the closing bracket or the argument of a known function), it will let you know that it wants more by means of a + sign. Give R what it wants to solve the problem.

  ```
  > install.packages(
  + )
  ```

- When you enter a path on Windows (e.g. `setwd("C:\CLSR")`), the backslash \ has a different meaning in R, which generates an error. You should use / (`setwd("C:/CLSR")`) or // (`setwd("C:\\CLSR")`) instead.
- In a script, all the lines of code are interdependent. Make sure you do not forget to store the relevant programming data in a named variable if you need to reuse it later on in your code.

2.14 Further Reading

- Albert and Rizzo (2012)
- Burns (2011)
- Crawley (2012)

Exercises

2.1. Vectors

a. Without using the R console, say what `z` contains after the following assignments:

```
> x <- c(3, 8)
> y <- 2
> z <- c(x,y)
```

```
> x <- c(3, 7, 9)
> y <- x[3]-x[2]
> z <- y + x[1]
```

b. Explain the error message that you obtain when you enter the following command:

```
> x <- c("3", "7", "9")
> x[1]+x[2]
```

c. Without using the R console, say what R outputs after the following two lines of code.

```
> y <- c(1,3,5,5)
> y[c(1,3)]
```

d. Without using the R console, say what `g` contains.

```
> i <- rep(1,5)
> j <- rep(6,7)
> k <- rep(i, 3)
> g <- c(i,j,k)
```

e. Without using the R console, give the type and length of each vector.

```
> ww <- 1+3
> xx <- c(ww,4)
> yy <- c(xx,8)
```

What contains `z`?

```
> z <- rep(yy,3)
```

f. Break down the complex vector `i` using smaller intermediate vectors.

```
> i <- rep(c(seq(1,3), mean(c(2,5,9))), 3)
```

g. Assign the names “a”, “b”, “c”, “d” and “e” to a sequence of integers from 1 to 5 with increment 1.

h. Create the following vectors.

1. `"a" "b" "c" "a" "b" "c"`
2. `TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE`

i. Without using the `nchar()` function, say how many characters the following vector contains.

```
> character_vector <- "R is great!"
```

2.2. Matrices

Tab. 2.5 is a matrix that displays the frequency distribution of *sure* and *hell* in *A as NP* in the British National Corpus (Desagulier 2016).

Table 2.5: Co-occurrence table for *sure* and *hell* in the BNC

	hell	other NPs
sure	33	17
other adjectives	65	3638

a. Enter this table into R in a matrix format. Include row names and column names.
b. Calculate the row sums and the column sums.
c. Calculate what percentage the frequency of *sure as hell* is of the total number of *A as NP* constructions.

2.3. Data frames

Tab. 2.6 displays the top five covarying collexemes of *A as NP* in the BNC according to the log-likelihood ratio score (also known as G^2). For example, the strongest attraction between the adjective and the NP in the construction is between *good* and *gold* in *good as gold* (Desagulier 2016).

Table 2.6: The top five pairs of covarying collexemes of *A as NP* in the BNC

rank	A	NP	G^2
1	*good*	*gold*	288.82
2	*quick*	*flash*	189.29
3	*right*	*rain*	175.98
4	*large*	*life*	164.55
5	*safe*	*houses*	148.32

a. Enter this data set into R in a data frame format. Make sure the values in the first column are treated as data points, not as row names.
b. Calculate the mean G^2 score of all five pairs.
c. Calculate the mean G^2 score of the top three pairs.
d. Sort the adjectives alphabetically in ascending order.
e. Sort the NPs alphabetically in descending order.

References

Albert, Jim, and Maria Rizzo. 2012. *R by Example*. Use R! New York: Springer.

Burns, Patrick. 2011. *The R Inferno*. http://www.burns-stat.com/pages/Tutor/R_inferno.pdf (visited on 08/26/2015).

Crawley, Michael. 2012. *The R Book*, 2nd ed. Chichester: Wiley.

Desagulier, Guillaume. 2016. A Lesson from Associative Learning: Asymmetry and Productivity in Multiple-Slot Constructions. *Corpus Linguistics and Linguistic Theory* 12 (2): 173–219.

Haspelmath, Martin. 2011. The Indeterminacy of Word Segmentation and the Nature of Morphology and Syntax. *Folia Linguistica* 45 (1): 31–80.

Chapter 3
Digital Corpora

Abstract A corpus is a digital text or collection of texts. After presenting a tentative typology of corpora, this chapter provides guidelines as to making your own corpora before presenting the characteristics of ready-made, annotated corpora.

3.1 A Short Typology

Because comprehensive typologies of existing corpora have been proposed elsewhere (e.g. McEnery and Hardie 2012, pp. 1–22; Gries 2009a, Sect. 2.1.2; Hunston 2002, pp. 14–16), I will just briefly summarize the key features that corpus typologies usually integrate.

A corpus may be general or specific. A general corpus aims at representing a whole language, regardless of a specific dialect or time span. Such is the case of the BNC or the COCA. A specific corpus is representative only of a particular span of time or variety. The Brown Corpus is specific because it is based on a selection of publications restricted to a time frame. The Bergen Corpus of London Teenage Language is also specific because it is restricted to spoken registers among London teenagers.

A corpus may be static or dynamic. A static corpus has a fixed size, which will not change. For example, the Corpus of Contemporary American English (1990–2015) started as a dynamic corpus and grew in size until its compiler stopped adding more text materials 2015. Although the BNC covers a span of time (from the 1980s to 1993), it was static from the moment it came out in 1993. A dynamic corpus is also known as a monitor corpus, i.e. a corpus designed to keep track of current changes in a language. The Bank of English The Bank of English (1980s–present), which has increased progressively since its creation in the 1980s, is a paragon example of a dynamic corpus.

A corpus may be synchronic or diachronic. A diachronic corpus is mostly used for the corpus-based study of language change. A diachronic corpus does not need to cover a wide time span. A synchronic corpus is mostly used for the study of a synchronic state of a language, including variation.

As you realize now, corpora come in a wide variety of flavors. To show you the broad diversity of corpora, I have carried out a tentative typology of seventeen of the most widely used corpora in English. These corpora are:

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_3

- The Bank of English (BoE)
- The British National Corpus (BNC)
- The Brown Corpus (BROWN)
- The English component of the Child Language Data Exchange System (CHILDES)
- The Corpus of Contemporary American English (COCA)
- The Corpus of Historical American English (COHA)
- The Bergen Corpus of London Teenage Language (COLT)
- enTenTen12
- The Freiburg-LOB Corpus of British English (FLOB)
- The Freiburg-Brown corpus of American English (FROWN)
- The Corpus of Global Web-Based English (GloWbE)
- Helsinki Corpus of English Texts (Helsinki)
- the British component of the International Corpus of English (ICE-GB)
- The Lampeter Corpus of Early Modern English Tracts (Lampeter)
- The Lancaster-Oslo/Bergen Corpus (LOB)
- The London-Lund Corpus of Spoken English (LLC)
- the Santa Barbara Corpus of Spoken American English (SBCAE)

Each corpus was described with respect to seven variables:

- the variety of English represented in the corpus (British, American, or international),
- whether the corpus is general or specific,
- whether the corpus is static or dynamic,
- whether the corpus is synchronic or diachronic,
- the format in which the data is stored (as text, as audio files, as video files, or as a combination of several formats),
- the mode (spoken, written, or both),
- the size of the corpus (from very small to largest).

The data is summarized in Tab. 3.1. The table itself is hardly interpretable because, although it is reasonably small, it contains too much information for the naked eye. To whet your appetite, let me anticipate on the clustering methods that you will discover in a later chapter. To explore the table and represent it graphically I used a method known as multiple correspondence analysis (MCA). MCA takes as input tables that contain nominal categorical variables, such as Tab. 3.1. It computes the distances between the variables and transposes those distances to a two-dimensional plot such as Fig. 3.1. Each row and each column of the table is thus represented as a point in the Euclidean space.

If all corpora had the same profile, they would cluster at the center of the plot, where the horizontal and the vertical axes intersect. Such is not the case as evidenced by how much the corpora are spread across the plot. Small corpora cluster to the left of the plot. Diachronic corpora concentrate in the top-left corner, whereas other synchronic, small corpora cluster in the bottom-left corner. Large, dynamic corpora of international English cluster in the top-right corner of the plot, whereas other large, general, static corpora cluster in the bottom-right corner. One tendency that the plot does not show, but which we can guess easily is that the most recent corpora cluster in the right part of the plot. This is an illustration of the abovementioned arms race with respect to size.

3.2 Corpus Compilation: Kennedy's Five Steps

Kennedy (1998, pp. 70–85) claims that five steps are necessary to compile a corpus. The first step is the corpus design. The design of a corpus depends on two factors: (a) the compiler's research goals, and (b)

Table 3.1: Seventeen corpora described by seven variables

corpus	variety	general vs. specific	static vs. dynamic	synchronic vs. diachronic	stored data format	mode	size
Bank of English	var: international	general	dynamic	synchronic	text	spoken + written	largest
BNC	var: GB	general	static	synchronic	text	spoken + written	large
BROWN	var: US	general	static	synchronic	text	written	small
CHILDES-English	var: international	specific	dynamic	synchronic	text + audio + video	spoken + written	large
COCA	var: US	general	static	synchronic	text	spoken + written	large
COHA	var: US	specific	static	diachronic	text	written	large
COLT	var: GB	specific	static	synchronic	text	spoken	very small
enTenTen12	var: international	general	static	synchronic	text	written	very large
FLOB	var: GB	general	static	synchronic	text	written	small
Frown	var: US	general	static	synchronic	text	written	small
GloWbE	var: international	general	static	synchronic	text	written	very large
Helsinki	var: GB	specific	static	diachronic	text	written	small
ICE-GB	var: GB	general	static	synchronic	text + audio	spoken + written	small
Lampeter	var: GB	specific	static	diachronic	text	written	small
LOB	var: GB	general	static	synchronic	text	written	small
London Lund Corpus	var: GB	general	static	synchronic	text	spoken	very small
SBCSAE	var: US	specific	static	synchronic	text + audio	spoken	very small

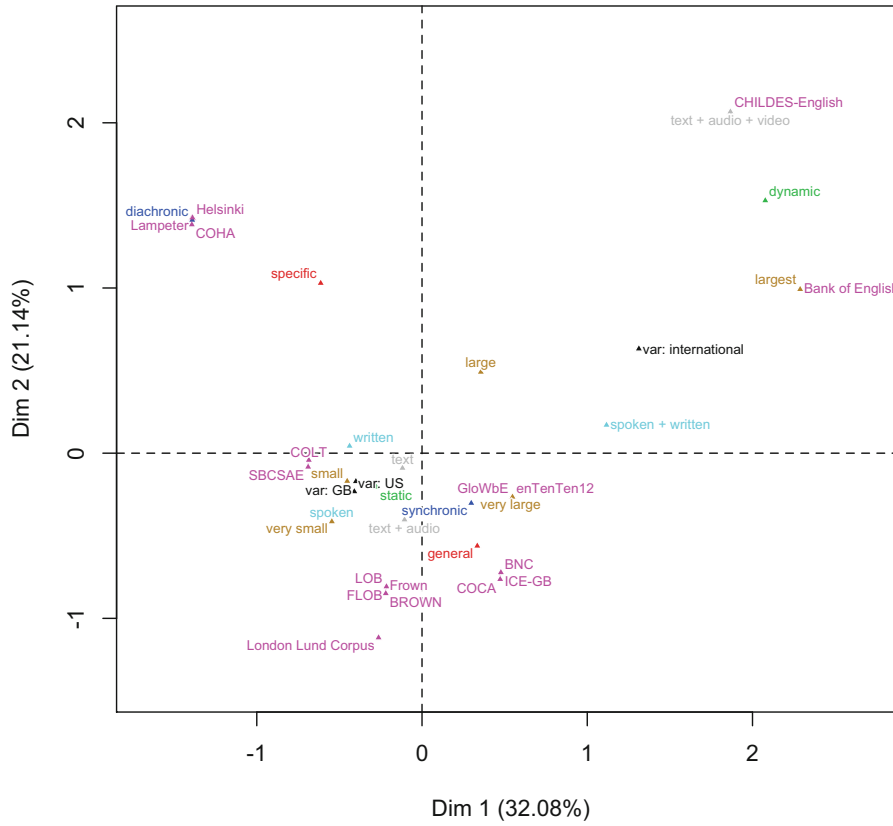


Fig. 3.1: A two-dimensional representation of corpora according to defining features (as obtained with multiple correspondence analysis)

whether the queries made from the corpus can be compared to the same queries made from another corpus. The research goals influence the choice of text types in the broad sense (e.g. spoken vs. written, synchronic vs. diachronic), content, architecture and size of the corpus. Designing a corpus is a long process that involves making one or several pilot corpora beforehand.

The second step is the planning of a neatly organized storage system. Generally, spoken corpora take up more space than written corpora because of the inclusion of sound files. In a not-so-remote past, most corpora were stored on CD-roms. Nowadays, they are stored on servers. Part of planning storage is the question of the metadata: whether they are part of the corpus files or placed in external metadata files is for you to decide. Finally, make sure that your architecture of folders and subfolders allows for efficient information retrieval.

The third step consists in obtaining permissions. Make sure that you have the legal right to use the texts that you include in your corpora. This is an essential step if you plan to share your corpus in whatever format you see fit. Be careful, copyright laws differ from one country to another.

The fourth step is text capture. For the compilation of a written corpus, you may have to manually transcribe handwritten documents, process text images (scanned text or pdf) via Optical Character Recognition (OCR). You may also be luckier and simply have to copy and paste character strings from an electronic document.

The fifth step is the choice of a markup convention, if your corpus is annotated. One such convention is the Text Encoding Initiative (TEI, http://www.tei-c.org/). It is originally based on the Standard Generalized Markup Language (SGML) and has integrated the more elaborate Extensible Markup Language (XML). The TEI is not limited to corpus compilation. It is a collection of norms used in the current effort to preserve texts in digital form. Its large tagset allows the user to indicate such text features as line numbers, line breaks, pauses, paragraphs, chapters, book parts, authors, speakers, date of recording, etc. The BNC is probably the best example of TEI implementation in corpus linguistics.

3.3 Unannotated Corpora

Any text can be used as a corpus as long as you can convert it into character strings. If your textual data are digital, the good news is that they are already encoded as character strings.

3.3.1 Collecting Textual Data

Collecting data is far less difficult than it used to be thanks to the Internet. Unless you are working on an endangered language, the amount of data that you can hope to get is not an issue. What is going to be worth serious thinking is selecting the kind of data that will help you answer your research question.

Project Gutenberg is a good place to start. It is a free online collection of more than 40000 books in electronic format in more than 40 languages.

3.3.2 Character Encoding Issues

Computer information is stored in units, such as bits, or bytes. A bit takes one of two values (0 and 1) and a byte equals eight bits. A byte can therefore encode $2^8 = 256$ possibilities. Text information is coded thanks to character sets. In a given character set, each character is assigned a digital profile. In other words, where we see a character, a computer sees a number, namely a string of digits for the representation of that character. To display and process a text correctly, it is necessary to know the character set that was used to encode the text.

Originally, most corpora were encoded in ASCII (American Standard Code for Information Interchange) because it was based on English characters, and because most corpus linguistics was done in English. The original ASCII table is encoded on seven bits. Therefore it has $2^7 = 128$ character possibilities. Each character is encoded on a single byte. Some characters that were not part of the original ASCII inventory, such as accented characters (é, è, ê, etc.), were encoded as "special" characters. That was not a viable approach as some non alphabetical character sets specific to written East Asian languages (Chinese, Japanese, and Korean) required far more "special" characters than single bytes could handle and were initially encoded with language-specific double-byte character sets. Whole sets were therefore designed to this aim: ISO 8859-1 (also known as latin1) for languages using the Latin alphabet in Western Europe, ISO/IEC 2022 for East Asian languages, KOI for languages written in Cyrillic script, etc.

Because modern corpus linguistics involves electronic data interchange in an increasingly globalizing environment, ASCII is being replaced by Unicode ("Unification Code"). Unicode has three main assets. First, it is platform-independent, which means it is usable on all computers, regardless of the operating system or locale. Second, it is not language-specific, which means that a single text may be multilingual (try processing a novel by James Joyce and you will see what I mean). Third, not only does Unicode define the numeric identity of each character, but it also handles how this value is translated into bits when the character is stored in a file or transferred electronically into another system. This is done via Unicode Transformation Formats (UTF). Among the three formats that have been proposed—UTF-32, UTF-16, and UTF-8—the latter is becoming the norm in corpus linguistics. One reason for this is that UTF-8 maximizes backward compatibility with ASCII. For more details on the assets of UTF-8 and why it is becoming a standard in text processing, see McEnery and Xiao (2005).

Despite efforts to unify character codes, encoding issues may occur when you load a corpus in the form of a character vector into R. The nature of these issues depends on how your operating system and its native encoding handle the native encoding of your corpus file. To minimize the risk of such encoding issues, it is recommended to set the locale to UTF-8. To know your locale, enter `Sys.getlocale()`.

```
> Sys.getlocale()
[1] "en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8"
```

On my Mac, the default language of the system if English, and the default encoding is UTF-8. Here, `LC` stands for "locale category". To obtain the list of all locale categories, and learn what they mean, enter `?locales`.

To change the locale settings, use the `Sys.setlocale()` function. Suppose the default language of your system is French and you want to set the locale to American English and UTF-8. You can do it in two steps.

```
> Sys.setlocale("LC_ALL", 'en_US.UTF-8') # step 1
[1] "en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8"
> Sys.setlocale("LC_MESSAGES", 'en_US.UTF-8') # step 2; might not work on Windows :-(
[1] "en_US.UTF-8"
> Sys.getlocale()
[1] "en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8"
```

For other languages, see Tab. 3.2.

Table 3.2: Languages and corresponding locale settings

language	locale setting
Chinese – Simplified	`zh_CN.UTF-8`
Chinese – Traditional	`zh_TW.UTF-8`
English – American	`en_US.UTF-8`
French	`fr_FR.UTF-8`
German	`de_DE.UTF-8`
Italian	`it_IT.UTF-8`
Japanese	`ja_JP.UTF-8`
Korean	`ko_KR.UTF-8`
Portuguese - Brazilian	`pt_BR.UTF-8`
Spanish	`es_ES.UTF-8`

Ideally, your corpus files should be encoded in UTF-8, in which case no issue should arise. In practice, there is some chance that you will process corpus files that were not originally encoded in UTF-8. What you should do is convert the corpus file from its native encoding into UTF-8. Let us see how this is done.

The file `hugo.txt` contains an excerpt from Victor Hugo's pamphlet *Napoléon le Petit*. The text is in French and natively encoded in latin1. When we load the file,

```
> hugo <- scan(file="C:/CLSR/chap3/hugo.txt", what="character", sep="\n") # Windows
> hugo <- scan(file="/CLSR/chap3/hugo.txt", what="character", sep="\n") # Mac
> head(hugo)
```

we see that the native encoding clashes with the locale, as evidenced by the strange rendition of accented characters.

```
[1] "Ce moment eut quelque chose de religieux. L'assembl\xe9e n'\xe9tait plus"
[2] "l'assembl\xe9e, c'\xe9tait un temple. Ce qui ajoutait \xe0 l'immense"
[3] "signification de ce serment, c'est qu'il \xe9tait le seul qui f\xfbt pr\xeat\xe9"
[4] "dans toute l'\xe9tendue du territoire de la r\xe9publique. F\xe9vrier avait"
[5] "aboli, avec raison, le serment politique, et la constitution, avec"
[6] "raison \xe9galement, n'avait conserv\xe9 que le serment du pr\xe9sident. Ce"
```

There are two solutions to this problem. The first solution is to convert the character vector into UTF-8 by using the `iconv()` function. The first argument is your character vector. The second argument is the native encoding of the text file. The third argument is the target encoding.

```
> hugo.utf8 <- iconv(hugo, from="latin1", to="UTF-8")
> head(hugo.utf8)
[1] "Ce moment eut quelque chose de religieux. L'assemblée n'était plus"
[2] "l'assemblée, c'était un temple. Ce qui ajoutait à l'immense"
[3] "signification de ce serment, c'est qu'il était le seul qui fût prêté"
[4] "dans toute l'étendue du territoire de la république. Février avait"
[5] "aboli, avec raison, le serment politique, et la constitution, avec"
[6] "raison également, n'avait conservé que le serment du président. Ce"
```

The second solution consists in declaring the native file encoding as you load the corpus file into a character vector.

```
> # Windows
> hugo.utf8.2 <- scan(file="C:/CLSR/chap3/hugo.txt", what="char", sep="\n", fileEncoding="latin1")
> # Mac
> hugo.utf8.2 <- scan(file="/CLSR/chap3/hugo.txt", what="char", sep="\n", fileEncoding="latin1")
> head(hugo.utf8.2)
```

R will do the rest, assuming the locale is set to UTF-8.

```
[1] "Ce moment eut quelque chose de religieux. L'assemblée n'était plus"
[2] "l'assemblée, c'était un temple. Ce qui ajoutait à l'immense"
[3] "signification de ce serment, c'est qu'il était le seul qui fût prêté"
[4] "dans toute l'étendue du territoire de la république. Février avait"
[5] "aboli, avec raison, le serment politique, et la constitution, avec"
[6] "raison également, n'avait conservé que le serment du président. Ce"
```

Note that in order to convert a corpus file, you must know its native encoding. To know what encodings R can convert from, enter `iconvlist()`.

Unfortunately, I cannot address all encoding issues here, for reasons of space. If you encounter such issues, given the high number of potential causes (especially on Windows), your safest bet is to search Stack Overflow preferably using the hashtags #R, #encoding, and #UTF-8 in the search box. If you are dealing with non-English, non-ASCII texts, I advise you to read Gries (2009a, excursus 4.5).

3.3.3 Creating an Unannotated Corpus

Creating a plain-text, unannotated corpora is not difficult in the digital age. If you plan on working from old manuscripts, as is trendy in the digital humanities, you may want to check digital archives before compiling your own corpus. For example, if you work on English from a diachronic perspective, Early English Books Online is worth considering (EEBO, `http://eebo.chadwyck.com/home`). If you work on French, the Gallica catalogue of the Bibliothèque Nationale de France is a good place to start (`http://gallica.bnf.fr/`). These resources are in full digital facsimile from microfilm or scanned collections. If you are lucky, the documents that you are interested in will be processed with OCR (Sect. 3.2). If not, it will be your job to do it if you intend to conduct a large-scale quantitative study. The bad news is that OCR works best with contemporary print characters in alphabetic languages and not so well with old or non-alphabetic fonts or handwriting. The good news is that OCR is offered all over the Web and in some software suites such as Adobe Acrobat.

Although much easier now than in the good old days, compiling a digital corpus should not dispense from paying due respect to the issues of representativeness (Sect. 1.2.1.2) and balance (Sect. 1.2.1.3). For this reason, it is primordial that you determine a sampling scheme that matches your objectives prior to collecting texts.

3.4 Annotated Corpora

A corpus is annotated when it consists of more than the actual words in the document (which is generally known as plain text). Annotation is a multilayered term that includes (but is not limited to) markup, POS-tagging, and semantic tagging.

3.4.1 Markup

Markup refers to the conventions used to annotate a corpus. With respect to written corpora, SGML and XML are the most widely used conventions (the norm is now XML). SGML and XML place tags between angled brackets < >. The tags follow an attribute-value format: e.g. `<teiHeader>...</teiHeader>` delimits the header of a BNC XML corpus file, and `<w ...>...</w>` delimits a word. For spoken corpora, such as the database of child language CHILDES, the CHAT format (Codes for the Human Analysis of Transcripts) is preferred.

Annotations can be metalinguistic and linguistic. Metalinguistic annotations—also known as structural markup—are generally placed in the header of the corpus file. Each corpus file in the XML edition of the BNC contains a TEI-compliant header that describes its contents whose details are described on the BNC homepage: http://www.natcorp.ox.ac.uk/docs/URG.xml?ID=cdifhd.

3.4.2 POS-Tagging

Above all, POS-tagging is used to annotate the words (and sometimes morphemes or more complex morpho-syntactic units) for grammatical categories in a text file (preposition, noun, adjective, verb, etc.). It is a prerequisite if you want to conduct large-scale yet in-depth queries.

Here is a simple example. The text below is POS-tagged using a simplified format. The tags are appended to the words (and punctuation).

```
Linguistics_NN1 makes_VVZ you_PPY want_VVI to_TO stay_VVI
up_RP late_RR at_II night_NNT1 and_CC gives_VVZ you_PPY
a_AT1 reason_NN1 to_TO wake_VVI up_RP in_II the_AT
morning_NNT1 ._.
```

As you can see, each word is assigned a tag. For example, *Linguistics* is a singular noun (`NN1`), *makes* is the *-s* form of the lexical verb (`VVZ`), *you* is the 2nd person personal pronoun (`PPY`), etc. The example below is a sentence from the BNC in plain text format and its equivalent with POS tags in the far more elaborate XML format.

a sentence from the British National Corpus

```
This virus affects the body's defence system so that it
cannot fight infection.
```

the same sentence, with POS tags (XML)

```
<s n="3"><w c5="DT0" hw="this" pos="ADJ">This </w><w c5="NN1"
hw="virus" pos="SUBST">virus </w><w c5="VVZ" hw="affect"
pos="VERB">affects </w><w c5="AT0" hw="the" pos="ART">the </
w><w c5="NN1" hw="body" pos="SUBST">body</w><w c5="POS" hw="'s"
pos="UNC">'s </w><w c5="NN1" hw="defence" pos="SUBST">defence </
w><w c5="NN1" hw="system" pos="SUBST">system </w><mw c5="CJS"><w
c5="AV0" hw="so" pos="ADV">so </w><w c5="CJT" hw="that"
pos="CONJ">that </w></mw><w c5="PNP" hw="it" pos="PRON">it </
w><w c5="VM0" hw="can" pos="VERB">can</w><w c5="XX0" hw="not"
pos="ADV">not </w><w c5="VVI" hw="fight" pos="VERB">fight </w><w
c5="NN1" hw="infection" pos="SUBST">infection</w><c c5="PUN">.</
c></s>
```

I will come back to the meaning of the attributes later on. Each annotation scheme comes with a tagset. The tagset used in the BNC XML is known as C5. It consists of about 60 tags and it is available here: http://www.natcorp.ox.ac.uk/docs/c5spec.html.[1]

3.4.3 POS-Tagging in R

POS-tagging is generally done by means of an automatic tagger such as CLAWS, the Constituent Likelihood Automatic Word-tagging System (Garside 1987),[2] the Stanford Log-linear Part-Of-Speech Tagger (Toutanova et al. 2003),[3] or the very powerful TreeTagger (Schmid 1994).[4]

Most POS taggers claim a very high accuracy (e.g. 96–97% for CLAWS). However, automatic tagging should always be reviewed afterwards. This step is made necessary by the potentially high number of ambiguous words, which automatic taggers might find hard to disambiguate (e.g. *pretty* as an adjective vs. *pretty* as an adverb).[5] Smith (1997) gives an example of the kind of sentence that the best programs find hard to tag:

(1) what's he want to prove?

The uncontracted equivalent of *what's* is *what does*. Yet, because most of the time *'s* is a present-tense enclitic form of the copula *be* in the third person of the singular, chances are that most taggers will get it wrong. To circumvent this issue, the CLAWS tagger has context-sensitive rules that correct previously tagged problematic cases. Despite this patch, there is a remaining 2% of errors.

[1] Its successor, C7, contains 140 tags.

[2] http://ucrel.lancs.ac.uk/claws/.

[3] http://nlp.stanford.edu/software/tagger.shtml.

[4] http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/.

[5] Although, nowadays, state-of-the-art probabilistic taggers would do quite a great job at disambiguating these uses in context.

Several R packages offer POS-tagging. I will focus on koRpus and openNLP, which are based on TreeTagger. Before using these packages, TreeTagger should be installed on your system. The good news is that TreeTagger is very efficient and supports a wide variety of languages. The bad news is that it is not as easy to install as an R package, depending on the distribution of your operating system. Having said that, TreeTagger is not that hard to install if you follow the instructions listed on its dedicated website. If you are a Windows user, instructions for download and install can be found here: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/#Windows. If you are a Mac or Linux user, just follow the download and install instructions at the top of Treetagger's homepage: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/. If you encounter issues, here are some more installation hints: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/installation-hints.txt.

Let us start with koRpus. First, create a file with some text in it for tagging, e.g. example.text.to.be.tagged.txt. Your text should be encoded in UTF-8. You can set the encoding with your text editor when you save the file or with R (see Sect. 3.3.2). Our example text will be doi.txt. Download, install, and load koRpus.

```
> rm(list=ls(all=TRUE))
> install.packages("koRpus")
> library(koRpus)
```

Define a pathname for the textfile that contains the text to annotate. In the example below, I created a subfolder named koRpus in /CLSR/chap3, hence the path /CLSR/chap3/koRpus/doi.txt.

```
> pathname <- "/CLSR/chap3/koRpus/doi.txt"
```

It is now time to perform POS-tagging with TreeTagger (which, at this stage, should be installed on your system). In the next line of code pay attention to the following: TT.options=list(path="/TreeTagger", preset="en")). In path="/TreeTagger", "/TreeTagger" is the path to my TreeTagger files. As you may have guessed from the path, I installed TreeTagger on my root drive. If you are a Windows user, the path should look like "C:/TreeTagger". Windows users should therefore enter something like path="C:/TreeTagger" if TreeTagger has been installed on the root drive. lang="en" means that the text you want to annotate is in English. In the TreeTagger world, each language has an ISO 639-1 code. The ISO 639-1 code is "en" for English, "fr" for French, "sv" for Swedish, "he" for Hebrew, etc. Here is a link to the full list of ISO codes: http://www-01.sil.org/iso639-3/codes.asp?order=639_1&letter=%25. Make sure TreeTagger handles the language that you are studying. Mind you, if you plan to annotate a text in a language other than English, download and install in your TreeTagger folder: (a) the parameter file (UTF-8) that corresponds to the language, and (b) the chunker parameter file (UTF-8) of your language. These files are available from http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/. Depending on the language, you may have to browse the web to find the parameter file and the chunker parameter file (e.g. Hebrew and Swedish). In any case, you need at least one parameter file per language. For example, if your text is in Spanish, download and install Spanish parameter file (UTF8) and, because the chunking function is also available for Spanish, download and install Spanish chunker parameter file. Both files should be placed unzipped in your TreeTagger folder. In TT.options=list(path="/

`TreeTagger", preset="en")),preset="en"` refers to the corpus that served for the training of TreeTagger. Here, TreeTagger has been trained on an English reference corpus.

```
> tagged.text <- treetag(pathname, treetagger="manual", lang="en",
+                        TT.options=list(path="/TreeTagger", preset="en"))
```

`tagged.text` is a complex object that contains all kinds of POS-related summary statistics regarding the input text. The output can be inspected by calling the `TT.res` object.

```
> tagged.text@TT.res
```

Thanks to this object, you can easily plot the distribution of the parts of speech in your text (Fig. 3.2).

```
> barplot(sort(table(tagged.text@TT.res$wclass), decreasing=TRUE,
+              main="distribution of word classes"))
```

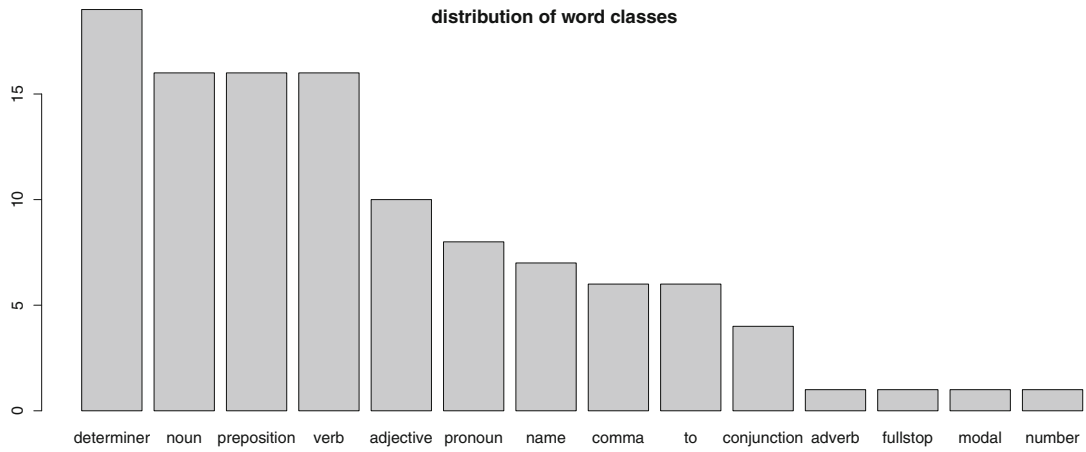


Fig. 3.2: The distribution of word classes according to `koRpus`

Although very useful as a tool for text analysis, `koRpus` does not readily give access to the tagged text. This is where a combination of `openNLP` and several NLP and text analysis packages step in. `openNLP` relies on the probabilistic Apache OpenNLP Maxent Part of Speech tagger and pre-trained models (one model for each language).[6] Using pre-trained models is fine as long as your text files do not show too much deviation (spelling- or register-wise) from the language on which the pre-training has been done. For your information, pre-training is based on contemporary news language.

POS-tagging requires a series of steps that may be slightly too ambitious if you are new to R. Feel free to come back to this code once your are more familiar with character string processing (Chap. 4). Fortunately,

[6] https://opennlp.apache.org/documentation/manual/opennlp.html.

Martin Schweinberger, from Hamburg University, Germany (http://www.martinschweinberger.de/blog/), has written a function that combines all these steps into a single function. Before you load the function, download and install the following packages: `stringr`, `NLP`, and `openNLP`.

```
> install.packages("NLP", "openNLP", "stringr")
```

You also need a pre-trained model. The model should be installed from a specific repository, namely http://datacube.wu.ac.at/. The suffix of the model's name is the language's ISO code. The models for Spanish, Italian, and Dutch end with `es`, `it`, and `nl` respectively. The model for English is named `openNLPmodels.en`. It should be installed as follows. You will not need to repeat this step.

```
> install.packages("openNLPmodels.en", repos = "http://datacube.wu.ac.at/", type = "source")
```

You may now enter the function.

```
> POStagging <- function(path = path) {
+   library(stringr)
+   library(NLP)
+   library(openNLP)
+   library(openNLPmodels.en)
+   corpus.files = list.files(path = path, pattern = NULL, all.files = T,
+                             full.names = T, recursive = T, ignore.case = T,
+                             include.dirs = T)
+   corpus.tmp <- lapply(corpus.files , function(x) {
+     scan(x, what = "char", sep = "\t", quiet = T) } )
+   corpus.tmp <- lapply(corpus.tmp, function(x){
+     x <- paste(x, collapse = " ") } )
+   corpus.tmp <- lapply(corpus.tmp, function(x) {
+     x <- enc2utf8(x) } )
+   corpus.tmp <- gsub(" {2,}", " ", corpus.tmp)
+   corpus.tmp <- str_trim(corpus.tmp, side = "both")
+   Corpus <- lapply(corpus.tmp, function(x){
+     x <- as.String(x) } )
+   sent_token_annotator <- Maxent_Sent_Token_Annotator()
+   word_token_annotator <- Maxent_Word_Token_Annotator()
+   pos_tag_annotator <- Maxent_POS_Tag_Annotator()
+   lapply(Corpus , function(x){
+     y1 <- annotate(x, list(sent_token_annotator ,
+                            word_token_annotator))
+     y2<- annotate(x, pos_tag_annotator , y1)
+     y2w <- subset(y2, type == "word")
+     tags <- sapply(y2w$features , "[[", "POS")
+     r1 <- sprintf("%s/%s", x[y2w], tags)
+     r2 <- paste(r1, collapse = " ")
+     return(r2) } )
+ }
```

The function `POStagging()` takes one argument: the path to the folder that contains your corpus file(s). The function will POS-tag all the text files in the folder. Windows users should enter

```
> POStagging(path = "C:/CLSR/chap3/koRpus/") # Windows
```

and Mac users should enter

```
> POStagging(path = "/CLSR/chap3/koRpus/") # Mac
```

Each word is assigned a POS tag after a slash.

```
When/WRB in/IN the/DT Course/NNP of/IN human/JJ events/
NNS it/PRP becomes/VBZ necessary/JJ for/IN one/CD
people/NNS to/TO dissolve/VB the/DT political/JJ bands/
NNS which/WDT have/VBP connected/VBN them/PRP with/IN
another/DT and/CC to/TO assume/VB among/IN the/DT powers/
NNS of/IN the/DT earth/NN ,/, the/DT separate/JJ and/CC
equal/JJ station/NN to/TO which/WDT the/DT Laws/NNP of/
IN Nature/NNP and/CC of/IN Natures/NNP God/NNP entitle/
VBP them/PRP ,/, a/DT decent/JJ respect/NN to/TO the/
DT opinions/NNS of/IN mankind/NN requires/VBZ that/IN
they/PRP should/MD declare/VB the/DT causes/NNS which/WDT
impel/VBP them/PRP to/TO the/DT separation/NN ./.  We/PRP
hold/VBP these/DT truths/NNS to/TO be/VB self-evident/JJ
,/, that/IN all/DT men/NNS are/VBP created/VBN equal/JJ
,/, that/IN they/PRP are/VBP endowed/VBN by/IN their/
PRP$ Creator/NN with/IN certain/JJ unalienable/JJ Rights/
NNS ,/, that/WDT among/IN these/DT are/VBP Life/NN ,/,
Liberty/NNP and/CC the/DT pursuit/NN of/IN Happiness/NNP
./.  (...)
```

The tags used by `openNLP` come from the tagset of the Penn English Treebank (Tab. 3.3).

3.4.4 Semantic Tagging

Semantic tagging is a very challenging task and certainly not one that will be resolved anytime soon, despite the recent advances in deep learning. Some semantic taggers exist, outside R, but they work only for some word classes and propose rather broad semantic categories.

The UCREL Semantic Analysis System (USAS)[7] is based on 21 semantic categories, some of which are subdivided into finer-grained discourse fields. Below is an example of a simple sentence—*Linguistics is an empirical science*—annotated thanks to the USAS online tagger[8]:

```
Linguistics_Q3 is_A3+ an_Z5 empirical_X2.4 science_Y1
._PUNC
```

A semantic tag is appended to each word. The USAS tagset[9] tells you what semantic category each word belongs to:

- `Q3`: "Language, speech and grammar";
- `A3`: "Being";
- `Z5`: "Grammatical bin";
- `X2.4`: "Investigate, examine, test, search";
- `Y1`: "Science and technology in general".

[7] http://ucrel.lancs.ac.uk/usas/.

[8] http://ucrel.lancs.ac.uk/usas/tagger.html.

[9] http://ucrel.lancs.ac.uk/usas/USASSemanticTagset.pdf.

Table 3.3: The University of Pennsylvania (Penn) Treebank Tagset

POS tag	POS category
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Although interesting as a first approach, this kind of semantic tagger is at bay when it comes to polysemous words. For instance, *hot* in *he was a rather hot seller* and in *it was a rather hot morning* will be assigned the same tag, namely `O4.6+` ("temperature +"). Semantic tagging is currently moving towards vector semantics (Mikolov et al. 2013a,b,c; Pennington et al. 2014).

3.5 Obtaining Corpora

Unless you need a corpus for a specific research purpose, chances are that you will find what you need among the vast inventory of available annotated corpora (Tab. 3.4). Here is a tentative list of publicly available POS-tagged corpora based on a survey conducted by Tobias Horsmann (University of Duisburg, Germany) on the Corpora mailing list in July 2016.[10]

Table 3.4: Free, publicly available, POS-tagged corpora

language	corpus name	size	URL
Coptic	Coptic scriptorium corpora	N/A (several corpora)	http://data.copticscriptorium.org/
Croatian	SETimes.HR Croatian dependency treebank	8000 sentences	http://nlp.ffzg.hr/resources/corpora/setimes-hr/
Danish	Copenhagen Dependency Treebank	100000 tokens	https://github.com/mbkromann/copenhagen-dependency-treebank
Dutch	The Alpino Treebank	150000 tokens	https://www.let.rug.nl/vannoord/trees/
English	The Georgetown University Multilayer Corpus	44079 tokens	https://corpling.uis.georgetown.edu/gum/
English	The NAIST-NTT Ted Talk Treebank	23158 tokens	http://ahclab.naist.jp/resource/tedtreebank/
Finnish	FinnTreeBank	162000 types (incl. punctuation)	http://www.ling.helsinki.fi/kieliteknologia/tutkimus/treebank/
French	Deep-sequoia	67038 tokens	https://deep-sequoia.inria.fr/corpus/
French	MULTITAG	370197 tokens	https://perso.limsi.fr/pap/free_multitag.tgz
German	TIGER corpus	900000 tokens	http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/tiger.html
German	The Hamburg Dependency Treebank	261821 sentences	https://corpora.uni-hamburg.de/drupal/en/islandora/object/treebank:hdt
Icelandic	Icelandic Parsed Historical Corpus	1002390 tokens	http://linguist.is/icelandic_treebank/Download
Icelandic	Tagged Icelandic Corpus	25000000 tokens	http://www.malfong.is/index.php?lang=en&pg=mim
Italian	Turin University Treebank	N/A (several corpora)	http://www.di.unito.it/~tutreeb/corpora.html
Italian	PAISÀ	250000000 tokens	http://www.corpusitaliano.it/
Norwegian	The Norwegian Dependency Treebank	300000 tokens	http://www.nb.no/sprakbanken/show?serial=sbr-10
Polish	The National Corpus of Polish (NKJP)	1000000 tokens	http://nkjp.pl/index.php?page=0&lang=1
Portuguese (Brazilian)	MAC-MORPHO	3013904 tokens	http://nilc.icmc.usp.br/macmorpho/
Russian	Russian Open Corpus	1344456 tokens	http://opencorpora.org/?page=downloads
Slovene–English	Slovene-English Parallel Corpus	1000000 tokens	http://nl.ijs.si/elan/
Spanish	IULA Spanish LSP Treebank	590000 tokens	https://www.iula.upf.edu/recurs01_tbk_uk.htm
Swedish	Swedish Treebank	350000 + 1200000 tokens	http://stp.lingfil.uu.se/%7Emojgan/UPDT.html

The Oxford Text Archive (http://ota.ox.ac.uk) lists resources that are available for free as long as you can prove that you belong to an academic institution and attest that you will not make a commercial use of the corpora. All versions of the British National Corpus are available from the OTA. In the remainder of this book, you will need the BNC Baby. Go to http://ota.ox.ac.uk/desc/2553 and click on "Download zip".

Exercise

3.1. POS-Tagging a Text in German

The text file `Brot.txt` contains a few sentences of contemporary German from the Wikipedia article on bread (https://de.wikipedia.org/wiki/Brot). The file can be found at the following location: `(C:)/CLSR/chap3/exercise/Brot.txt`. Your task is to POS-tag the text.

The code chunk below is a generic function (`tagPOS()`) for POS-tagging with TreeTagger. Use the function to POS tag `Brot.txt`.

```
> library(NLP)
> library(openNLP)
```

[10] http://clu.uni.no/corpora/.

```
>
> tagPOS <-  function(x, ...) {
+   s <- as.String(x)
+   word_token_annotator <- Maxent_Word_Token_Annotator(language="de", probs=FALSE, model=NULL)
+   a2 <- Annotation(1L, "sentence", 1L, nchar(s))
+   a2 <- annotate(s, word_token_annotator, a2)
+   a3 <- annotate(s, Maxent_POS_Tag_Annotator(language="de", probs=FALSE, model=NULL), a2)
+   a3w <- a3[a3$type == "word"]
+   POStags <- unlist(lapply(a3w$features, `[[`, "POS"))
+   POStagged <- paste(sprintf("%s_%s", s[a3w], POStags), collapse = " ")
+   list(POStagged = POStagged, POStags = POStags)
+ }
```

Before running the code and using the function, you will need to:

- install the required parameter files for German from http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/;
- install and load the German model's name as a package for R from http://datacube.wu.ac.at;
- add some arguments to

  `Maxent_Word_Token_Annotator()` and
  `Maxent_POS_Tag_Annotator()`

  so that TreeTagger understands that the text to process is in German and does not use the default settings for English.

References

Davies, Mark. 1990–2015. *The Corpus of Contemporary American English: 450 Million Words, 1990–Present*. http://corpus.byu.edu/coca/.

Garside, Roger. 1987. The CLAWS Word-Tagging System. In *The Computational Analysis of English: A Corpus-Based Approach*, ed. Roger Garside, Geoffrey Leech, and Geoffrey Sampson, 30–41. London: Longman.

Gries, Stefan Thomas. 2009. *Quantitative Corpus linguistics with R: A Practical Introduction*. New York, NY: Routledge.

Hunston, Susan. 2002. *Corpora in Applied Linguistics*. Cambridge: Cambridge University Press.

Kennedy, Graeme. 1998. *An Introduction to Corpus Linguistics*. Harlow: Longman.

McEnery, Tony, and Andrew Hardie. 2012. *Corpus Linguistics : Method, Theory and Practice*. Cambridge Textbooks in Linguistics. Cambridge, New York: Cambridge University Press.

McEnery, Tony, and Richard Xiao. 2005. Character Encoding in Corpus Construction. In *Developing Linguistic Corpora: A Guide to Good Practice*, ed. Martin Wynne, 47–58. Oxford: Oxbow Books.

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig. 2013a. Linguistic Regularities in Continuous Space Word Representations. In *Proceedings of NAACL-HLT*, 746–751. http://www.aclweb.org/anthology/N/N13/N13-1090.pdf.

Mikolov, Tomas, et al. 2013b. Distributed Representations of Words and Phrases and Their Compositionality. CoRR abs/1310.4546. http://arxiv.org/abs/1310.4546.

Mikolov, Tomas, et al. 2013c. Efficient Estimation of Word Representations in Vector Space. CoRR abs/1301.3781. http://arxiv.org/abs/1301.3781.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. Glo Ve: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 1532–1543. http://www.aclweb.org/anthology/D141162.

Schmid, Helmut. 1994. Probabilistic Part-of-Speech Tagging Using Decision Trees. In *International Conference on New Methods in Language Processing*, Manchester, UK, 44–49. http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger1.pdf.

Smith, Nicholas. 1997. Improving a Tagger. In *Corpus Annotation: Linguistic Information from Computer Text Corpora*, ed. Roger Garside, Geoffrey N. Leech, and Tony McEnery, 137–150. London: Longman.

The Bank of English. 1980s–present. http://www.titania.bham.ac.uk/. Jointly owned by HarperCollins Publishers and the University of Birmingham.

Toutanova, Kristina, et al. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of HLT-NAACL 2003*, 252–259.

Chapter 4
Processing and Manipulating Character Strings

Abstract In this chapter, you will learn techniques to handle text material with R. Some of these techniques involve regular expressions, i.e. patterns that describe a set of strings. After presenting individual functions, this chapter teaches you how to combine them.

4.1 Introduction

Typical R users come from the hard sciences and seldom think of character strings as data that are worth analyzing. Their basic material is numbers or anything that can be converted into numeric values. Character strings are secondary because they involve nothing more than labels such as names for observations, variable, units, bins, etc. Admittedly, only a handful of basic functions are needed to edit labels in a data set.

In contrast, because their main material is textual, linguists appreciate R for its powerful character-string processing features. It is one thing to edit labels sporadically, but another to load vast quantities of texts and process them in a systematic way to be able to observe linguistic tendencies.

After reading this chapter, you will hopefully have all the linguist's tools you need to handle and process character strings with R. These tools are a prerequisite for what the next chapter will cover, namely tabulating and quantifying your findings.

4.2 Character Strings

I assume that you are already familiar with character strings as they were introduced in from Chap. 2. The paragraphs below examine the properties of character strings in more details and discuss some useful technicalities.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_4

4.2.1 Definition

As you already know, R objects can handle three main kinds of data: character, numeric, and logical. Corpora are made of text strings. The R objects that contain such strings belong to the character class. You recognize a character string in R thanks to the presence of quotes. To create a character vector, make sure to include them.

```
> a_numeric_vector <- 3.14
> class(a_numeric_vector)
[1] "numeric"
> from_numeric_to_character <- as.character(a_numeric_vector)
> class(from_numeric_to_character)
[1] "character"
```

Once converted into a character vector, `a_numeric_vector` displays `"3.14"` between quotes. You can no longer use that number for calculation.

4.2.2 Loading Several Text Files

Section 2.9.1.6 explains how to load a single text file into a character vector. For the sake of illustration, let us load `whitman.txt` from your `CLSR/chap4` folder. The text file contains Walt Whitman's poem "When I heard the learn'd astronomer".

```
> whitman <- scan("C:/CLSR/chap4/whitman.txt", what="char", sep="\n") # Windows
> whitman <- scan("/CLSR/chap4/whitman.txt", what="char", sep="\n") # Mac
> head(whitman) # inspect
```

Because you have set `sep` to a new line (`"\n"`), each line is an element of the character vector `whitman`.

Because most corpora consist of several files, you need to know how to load several files into a character vector. For example, the BNC (XML Edition) consists of 4049 text files. If R runs on Windows, this is pretty easy. All you need to do open an interactive window with `choose.files()` and select all the text files you need in the corpus directory. If R runs on Mac OS, you need to embed `scan()` in two other functions: `dir()` and `select.list()`.

```
> corpus.files<-select.list(dir(scan(nmax=1, what=character(0)), full.names=TRUE), multiple=TRUE)
```

Then, assuming that all of them are in the same directory, enter the path to said directory.

Another option that works for both Windows and Unix-based operating systems is to use `list.files()`. This function outputs a character vector of the file names in a given directory. Its first argument is the path to the directory containing the text files. Three more arguments may be used: `pattern`, `all.files`, and `full.names`. Thanks to the `pattern` argument, you can use a regular expression to retrieve those file names that match the regular expression. For example, if your directory contains corpus files in `.txt` and `.xml` format and you are exclusively interested in the latter, `pattern="\\.xml"` will make sure that only those files that end with the `.xml` extension are retrieved. The `all.files` argument has to be set to `TRUE` if you want to retrieve all the corpus files in the corpus directory. Finally, the `full.names` argument has to be set to `TRUE` if you want the full path name of the corpus files.

4.3 First Forays into Character String Processing

It is now time to learn how to do the most basic tasks in character string processing: splitting, matching, and replacing. In this section, you will familiarize yourself with the three corresponding functions to split, match, and replace, namely `strsplit()`, `grep()`, and `gsub()`. In the next section, we will return to those functions and see how powerful they are once combined with regular expressions.

4.3.1 Splitting

You already know from Chap. 2 (Sect. 2.9.1.3) how to merge several vectors of one element or one vector of several elements into one vector of one element with `paste()`. Yet, you often need to do the exact opposite, that is to say split the corpus vector into as many elements as there are words. This comes in handy to know how many words your corpus sample consists of and to make frequency lists. R has a built-in function for splitting character strings: `strsplit()`. Its simplified argument structure is the following.

```
> strsplit(x, split)
```

The first argument `x` is the character vector to be split, and the second argument specifies what you want to split the character vector with.

Given the vector `r3`, the simplest way to extract the words that make up the sentence is to split at each space.

```
> richard3 <- "now is the winter of our discontent"
> strsplit(richard3, " ") # the second argument specifies the space
[[1]]
[1] "now"        "is"         "the"
[4] "winter"     "of"         "our"
[7] "discontent"
```

As evidenced by the double bracketing prefixing the output, `strsplit()` outputs a list. To obtain a vector containing the words, just prefix the function call with `unlist()`:

```
> unlist(strsplit(richard3, " "))
[1] "now"        "is"         "the"
[4] "winter"     "of"         "our"
[7] "discontent"
```

Counting the number of words in the string is now easy as pie, just like counting the number of characters per words.[1]

```
> length(unlist(strsplit(richard3, " "))) # number of words
[1] 7
> nchar(unlist(strsplit(richard3, " "))) # number of characters per words
[1]  3  2  3  6  2  3 10
```

[1] Note that where we see “words”, R still sees character strings.

4.3.2 Matching

Another basic character string operation is to find strings. In R parlance, "finding as string" is called "matching a string". This is done with the `grep()` function, whose simplified argument structure is the following.

```
> grep(pattern, x, ignore.case = FALSE, value = FALSE)
```

The first argument (`pattern`) is the character string you want to match. The second argument `x` is the character vector where the matches are sought. The third argument `ignore.case` specifies whether the pattern matching is case sensitive or not (it is case sensitive by default). The fourth argument specifies whether grep outputs the position of the match in the vector (`value` is set to `FALSE` by default) or the vector element containing the match (`value=TRUE`).

The vector `grep.ex` consists of three sentence. Each sentence is an element of the vector.

```
> grep.ex <- c("some linguists like Python", "some linguists prefer R", "some linguists like both")
> grep.ex
[1] "some linguists like Python"
[2] "some linguists prefer R"
[3] "some linguists like both"
```

Compare:

```
> grep("both", grep.ex)
[1] 3
> grep("both", grep.ex, value=TRUE)
[1] "some linguists like both"
```

The first instance of `grep()` returns `3`, which is the position of the vector element containing the match: `"both"` is the third vector element. The second instance of `grep()` returns the vector element that corresponds to the match.

4.3.3 Replacing and Deleting

The functions `sub()` and `gsub()` functions search and replace character strings with something else. Their respective simplified argument structures are the following:

```
> sub(pattern, replacement, x, ignore.case = FALSE)
> gsub(pattern, replacement, x, ignore.case= FALSE)
```

The first argument is the pattern you search for, the second argument is what you want to replace it with, and the third argument specifies whether the search is case sensitive (the search is case sensitive by default).

In the vector `r_and_j`, we want to match "Romeo" and replace it with "Superman" (sorry, Shakespeare).

```
> r_and_j <- "O Romeo, Romeo, wherefore art thou Romeo?"
> sub("Romeo", "Superman", r_and_j, ignore.case = FALSE)
[1] "O Superman, Romeo, wherefore art thou Romeo?"
> gsub("Romeo", "Superman", r_and_j, ignore.case = FALSE)
[1] "O Superman, Superman, wherefore art thou Superman?"
```

As you can see, `sub()` matches and replaces only the first instance of `Romeo`, whereas `gsub()` matches and replaces all instances.[2] For this obvious reason, `sub()` is hardly ever used by corpus linguists.

[2] The `g` in `gsub()` stands for "global".

If you set `ignore.case = TRUE`, then you do not need to care about uppercase or lowercase:

```
> gsub("romeo", "Superman", r_and_j, ignore.case = TRUE)
[1] "O Superman, Superman, wherefore art thou Superman?"
> gsub("ROMEO", "Superman", r_and_j, ignore.case = TRUE)
[1] "O Superman, Superman, wherefore art thou Superman?"
```

In R, you do not delete anything per se. Instead, you replace what you want to delete with nothing, or more precisely with an empty string. If you want to delete the commas in `r_and_j`, just replace `","` with `""`.

```
> gsub(",", "", r_and_j)
[1] "O Romeo Romeo wherefore art thou Romeo?"
```

4.3.4 Limitations

The above is just the tip of the iceberg with respect to what `grep()` and `gsub()` can do. In fact, these functions soon reach their limits if they are not combined with regular expressions.

Suppose you want to match all verbs ending in *'d* in your `whitman` vector. This won't work unless you find a way to tell R to look for a sequence of characters ending in *'d*. The only way to do it is to translate "a sequence of characters ending in *'d*" by means of a regular expression.

Suppose you now want to delete all punctuation except the apostrophe before making a frequency list. One option is to look for every type of punctuation mark (the comma, the colon, the semi-colon, etc.), with one line of code per punctuation type. A second, far more efficient method is to use one regular expression that covers all punctuation types.

4.4 Regular Expressions

Regular expressions are patterns that perform the same functions but in a more powerful way. Suppose you want to retrieve all proper names from a document. In a commercial word processor, you will not be able to do it unless you find a way of telling the processor to match all words that start with a capital letter, ignoring those words that appear in sentence-initial position. The easiest way to do it is to use a regular expression, i.e. a pattern that abstracts away from the specificities of each proper name and retains only what these names have in common: a capital letter followed by several word characters. If you do not use a regular expression, several steps are needed. If you use a regular expression, one step is enough.[3]

4.4.1 Overview

Regular expressions (also known as regexes) are not specific to R. They are found in all computer languages. In fact, even though R has all the regexes you need, other scripting languages such as Perl or Python offer a wider range of possibilities. Good text editors such as Notepad++ or TextWrangler understand regular expressions.

[3] Of course, you will have to deal with proper names that appear in sentence-initial position.

Concretely, regexes are expressions that combine characters to form a pattern. The pattern is a generic expression denoting one or several character strings to be matched.

Resources on regular expressions

- Regular expressions in R:
  `?regex`
  https://stat.ethz.ch/R-manual/R-devel/library/base/html/regex.html
- Regular expressions in general:
  http://www.regular-expressions.info/
  http://regexone.com/
  http://www.rexegg.com/

4.4.2 Literals vs. Metacharacters

A literal is a character that receives a normal, literal interpretation when it is part of a regular expression. For example, the pattern `"5"` matches the digit character "5" and the pattern `"dog"` matches the character string "dog".

On the other hand, a metacharacter is a character that receives a special interpretation in a regular expression. If you want a metacharacter to be used literally, you need to escape it. In most scripting languages, escaping is done with one backslash `\`. However, in R, you have to double the backslash: `\\`.

For example, the dollar sign `$` does not match the monetary unit but, as you will learn below (Sect. 4.4.3), the end of a string. The vector `dollar` has three elements. The dollar sign matches all three elements even though the currency symbol is present only in the first two. This is because all three elements have a string end, each of which is matched.

```
> dollar <- c("I paid $15 for this book.", "they received $50,000 in grant money", "two dollars")
> grep("$", dollar)
[1] 1 2 3
```

If you want to match the currency symbol, you need to escape it.

```
> grep("\\$", dollar)
[1] 1 2
```

Tab. 4.1 lists some characters that need escaping to be used literally. The sections below will explain what each metacharacter means.

4.4.3 Line Anchors

The line anchors `^` and `$` match the beginning and the end of a string respectively. For example, the regular expression `"sh"` will match the character `"s"`, followed by the character `"h"`. The regular expression

Table 4.1: Some metacharacters and how to escape them

name	metacharacter	how to escape it
dot	`.`	`\\.`
pipe	`|`	`\\\|`
opening bracket	`(`	`\\(`
closing bracket	`)`	`\\)`
opening square bracket	`[`	`\\[`
closing square bracket	`]`	`\\]`
opening brace or curly bracket	`{`	`\\{`
closing brace or curly bracket	`}`	`\\}`
caret or hat	`^`	`\\^`
dollar sign	`$`	`\\$`
asterisk	`*`	`\\*`
plus sign	`+`	`\\+`
question mark	`?`	`\\?`

`"^sh"` will match the same sequence of characters, but only at the beginning of a string, and `"sh$"` will match it at the end of a string.

```
> grep("sh", c("ashes", "shark", "bash"), value=TRUE) # matches all three vector elements
[1] "ashes" "shark" "bash"
> grep("^sh", c("ashes", "shark", "bash"), value=TRUE) # matches only "shark"
[1] "shark"
> grep("sh$", c("ashes", "shark", "bash"), value=TRUE) # matches only "bash"
[1] "bash"
```

Line anchors do not consume characters because their length is null.

4.4.4 Quantifiers

Quantifiers are placed right after a regular expression. They are meant to repeat the regular expression a given number of times. Tab. 4.2 lists the quantifying expressions available in R.

The vector `quant.ex` contains five elements.

```
> quant.ex <- c("gaffe", "chafe", "chalk", "encyclopaedia", "encyclopedia")
```

You want to match both `"gaffe"` and `"chafe"` (and only these two character strings) in one go. This is fairly easy because `"gaffe"` and `"chafe"` are the only strings that contain the `"f"` character. Because `"chafe"` has one `"f"` and `"gaffe"` has two, all you need to do is look for a character string that contains at least one letter `"f"`. Using quantifiers, this can be done in several ways.

```
> grep("f+", quant.ex, value=TRUE) # one or more "f"
[1] "gaffe" "chafe"
> grep("f{1,2}", quant.ex, value=TRUE) # one or two "f"
[1] "gaffe" "chafe"
> grep("f{1,}", quant.ex, value=TRUE) # at least one "f"
[1] "gaffe" "chafe"
```

Table 4.2: Quantifiers in R

quantifier	interpretation
*	zero or more occurrence(s) of the preceding regex
+	one or more occurrence(s) of the preceding regex
?	zero or one occurrence of the preceding regex
{n}	*n* occurrence(s) of the preceding regex
{n,}	at least *n* occurrences of the preceding regex
{,n}	up to *n* occurrences of the preceding regex
{n,m}	between *n* and *m* occurrences of the preceding regex

The same vector contains both the British and American spellings "encyclopaedia" and "encyclopedia". We want to retrieve both variants. Both spelling are identical except for one difference: "ae" vs. "e". The British spelling has one letter that the American spelling does not have. All we need to do is make the letter "a" optional in the regex by means of a quantifier. The most obvious option is to match a string that contains the word character "p" followed by zero or one instance of the word character "a", followed by one "e". Including "p" guarantees that the expression will not match any other string of the vector because "gaffe", "chafe", and "chalk" do not contain a "p". The expression can be formulated in several ways, using equivalent quantifying expressions.

```
> grep("pa?e", quant.ex, value=TRUE) # "p" followed by zero or one "a", followed by "e"
[1] "encyclopaedia" "encyclopedia"
> grep("pa{0,1}e", quant.ex, value=TRUE) # "p" followed by zero or one "a", followed by "e"
[1] "encyclopaedia" "encyclopedia"
> grep("pa*e", quant.ex, value=TRUE) # "p" followed by zero or more "a", followed by "e"
[1] "encyclopaedia" "encyclopedia"
```

4.4.5 Alternations and Groupings

The vector some.verbs contains some random verbs. For two of them, the British and American spelling variants are included.

```
> some.verbs <- c("italicize", "highlight", "italicise", "emphasise", "underline",
+             "emphasize", "insert", "place")
```

Suppose that, for the purpose of a study, you want to retrieve only those verbs that display a spelling alternation with respect to the *-ize* vs. *-ise* ending. You must tell R to look for character strings that end in "ise" or "ize". Because the endings are identical except for the *"s"/"z"* contrast, just tell R to match character strings whose endings alternate between "z" and "s".

```
> grep("i[sz]e", some.verbs, value=TRUE)
[1] "italicize" "italicise" "emphasise"
[4] "emphasize"
```

Square brackets subdivide a regular expression into alternative characters. For instance, the regular expression [abcd] means "a or b or c, or d".

In R, the pipe symbol (|) subdivides a regular expression not just into alternative characters but into alternative *patterns*. For instance, the regular expression `"oo|ee"` means a sequence of two `o` characters or a sequence of two `e` characters. In the vector `some.words`, the regular expression matches `foot`, `food`, and `feet`.

```
> some.words <- c("foot", "food", "fork", "flask", "feet")
> grep("oo|ee", some.words, value=TRUE)
[1] "foot" "food" "feet"
```

Sometimes, you need to embed alternating subpatterns *within* a regular expression. In the vector `verb.forms` below, you want to match the *-ing* forms of the verbs, namely `being`, `having`, and `doing`. All three verb forms have the *-ing* ending in common. The alternative patterns are the character strings that precede the ending: `be-`, `hav-`, and `do-`. These alternative patterns must be grouped within brackets.

```
> verb.forms <- c("do", "been", "had", "does", "did", "being",
+                 "done", "has", "were", "doing", "was", "having")
> grep("(be|hav|do)ing", verb.forms, value=TRUE)
[1] "being"  "doing"  "having"
```

The regex `"(be|hav|do)ing"` matches the string `"be"` or `"hav"` or `"do"` followed by the string `"ing"`. If you omit the brackets, the regex matches vector elements that contain `be`, `hav`, or `doing`, which is not what you want.

```
> grep("be|hav|doing", verb.forms, value=TRUE)
[1] "been"   "being"  "doing"  "having"
```

This is because the match encompasses all that appears on either side of the pipe symbol. Brackets are needed to delimit the alternating patterns.

4.4.6 Character Classes

In R, all alphanumeric characters are defined as `[a-zA-Z]`, which can be interpreted as any lowercase or uppercase letter. Any lowercase letter can be obtained with `[a-z]` or `letters`. Any uppercase letter can be obtained with `[A-Z]` or `LETTERS`. Common character ranges can also be specified using special character sequences (Tab. 4.3).

To make each character class explicit, let us use an example sentence and replace each class in turn with an asterisk.

```
> ex.sentence <- "Act 3, scene 1. To be, or not to be, that is the Question:"
```

When we replace `"."` with `"*"`, all characters are replaced, including spaces and punctuation.

```
> gsub(".", "*", ex.sentence)
[1] "**********************************************************"
```

When we replace `"\\w"` with `"*"`, all word characters are replaced. As expected, spaces and punctuation are not replaced.

```
> gsub("\\w", "*", ex.sentence)
[1] "*** *, ***** *. ** **, ** *** ** **, **** ** *** ********:"
```

Table 4.3: Essential character sequences in R

anchor	interpretation
.	any character except a newline
\\w	any word character
\\W	anything but a word character
\\d	any digit character
\\D	anything but a digit character
\\b	a word boundary
\\B	anything but a word boundary
\\s	any space character
\\S	anything but a space character

When we replace "\\W" with "*", only non-word characters are replaced. In the vector, this concerns spaces and punctuation.

```
> gsub("\\W", "*", ex.sentence)
[1] "Act*3**scene*1**To*be**or*not*to*be**that*is*the*Question*"
```

When we replace "\\d" with "*", only the two digit characters in the vector are replaced. When the substitution involves "\\D", it is the other way around.

```
> gsub("\\d", "*", ex.sentence)
[1] "Act *, scene *. To be, or not to be, that is the Question:"
> gsub("\\D", "*", ex.sentence)
[1] "****3********1*****************************************"
```

When we replace "\\b" with "*", all word boundaries are replaced. An asterisk marks the beginning and the end of each word. Pay attention to the string "be*,". The word boundary is after "be" and before the comma. As you can see, word boundaries have no length.

```
> gsub("\\b", "*", ex.sentence, perl=TRUE)
[1] "*Act* *3*, *scene* *1*. *To* *be*, *or* *not* *to* *be*, *that* *is* *the* *Question*:"
```

Like word boundaries, non-word boundaries have no length either. When we replace "\\B" with "*", an asterisk is inserted between word characters.

```
> gsub("\\B", "*", ex.sentence)
[1] "A*ct 3,* s*ce*ne 1.* T*o b*e,* o*r n*ot t*o b*e,* t*ha*t i*s t*he Q*ue*st*io*n:"
```

When "\\s" is replaced with "*", an asterisk is substituted for each space.

```
> gsub("\\s", "*", ex.sentence)
[1] "Act*3,*scene*1.*To*be,*or*not*to*be,*that*is*the*Question:"
```

When "\\S" is replaced with "*", asterisks are substituted for all characters, except space characters.

```
> gsub("\\S", "*", ex.sentence)
[1] "*** ** ***** ** ** *** ** *** ** *** **** ** *** *********"
```

Other common character ranges can be specified using character sequences of the form [:keyword:], as shown in Tab. 4.4. The inner brackets indicate a class name, whereas the outer brackets indicate a range. These character classes are based on the POSIX family of standards, POSIX being an acronym for the Portable Operating System Interface.

Table 4.4: Essential POSIX character classes

character class	what it means
`[[:alpha:]]`	alphabetic characters
`[[:digit:]]`	digits
`[[:punct:]]`	punctuation
`[[:space:]]`	space, tab, newline, etc.
`[[:lower:]]`	lowercase alphabetic characters
`[[:upper:]]`	uppercase alphabetic characters

4.4.7 Lazy vs. Greedy Matching

As you saw in Chap. 3, annotated corpus files contain tags delimited by angle brackets `< >` or appended to each word by means of an underscore symbol (_). The character vector `annotated.v` contains an annotated corpus sentence. Each word is tagged in a XML fashion, following the CLAWS5 tagset.

```
annotated.v <- "<w c5=\"DT0\" hw=\"this\" pos=\"ADJ\">this </w><w c5=\"VBZ\" hw=\"be\"
pos=\"VERB\">is </w><w c5=\"AT0\" hw=\"an\" pos=\"ART\">an </w><w c5=\"AJ0\" hw=\"annotated\"
pos=\"ADJ\">annotated </w><w c5=\"NN1\"  hw=\"sentence\" pos=\"SUBST\">sentence</w><c
c5=\"PUN\">."
```

Each item is marked by a start tag and a closing tag. The start tag contains:

- a code letter for the nature of the item (`w` if it is a word, `c` if it is a punctuation mark);
- a POS tag based on the CLAWS5 tagset;
- the head word (`hw`), or lemma;
- the value of the CLAWS5 wordclass.

The closing tag signals the end of a word unit.

While tags are useful for POS-based corpus queries, they are generally not required for lexeme-based frequency lists and should therefore be removed after use. You already know that there is no function dedicated to deleting, but that `gsub()` does the job quite well by replacing what is to be deleted with an empty string. The difficulty here is that we want to delete both the brackets and what is inside them. The problem is that what is inside each brackets changes from an item to the next.

If we apply what we have learned above about quantifiers and character classes, we might think that `<.*>` would do the trick. The expression means: look for an opening bracket, followed by zero or more characters except a newline, and a closing bracket. Yet look what happens when we substitute that expression for an empty string.

```
> gsub("<.*>", "", annotated.v)
[1] "."
```

By default, R matches the first opening bracket (`<w c5=\"DT0\"`) and then chooses the longest possible match, i.e. up to the last closing bracket in the vector (`c5=\"PUN\">`). This is known as greedy matching.

We must instead tell R to match an opening bracket until it finds the nearest closing bracket. To do this, just add a question mark after the regex for greedy matching: `.*?`.

```
> gsub ("<.*?>" , "" , annotated.v)
[1] "this is an annotated sentence."
```

This procedure is known as lazy matching. R iterates until it can no longer match an occurrence of an opening bracket.

4.4.8 Backreference

Backreference allows you to match a substring (i.e. a part of a character string), save it, index it, and do whatever you want with it. In the toy vector `letters.digits`, letters from `a` to `i` are paired with their respective positions in the alphabet. Each pair is separated by a space. Suppose you want to switch letters and positions. What you need to do is: (a) match each letter, (b) match each digit, (c) save both, (d) switch them. Backreferencing will do the trick in one go.

```
> letters.digits <- "a1 b2 c3 d4 e5 f6 g7 h8 i9"
> gsub("(\\w)(\\d)", "\\2\\1", letters.digits)
[1] "1a 2b 3c 4d 5e 6f 7g 8h 9i"
```

The regular expression in the first argument of `gsub()`—`"(\\w) (\\d)"`—matches any word character followed by any digit character. The first pair of brackets tells R to store `\\w` in memory. The second pair of brackets tells R to do the same for `\\d`. Each set of brackets is numbered according to the order of opening brackets. Therefore, the first pair is numbered `1` and the second pair is numbered `2`. This makes what is matched between the brackets available for further processing, providing you escape the numbers as follows: `\\1` and `\\2`. In the replacement argument of `gsub()`, the regular expression `"\\2\\1"` instructs R to switch letters and digits for each letter–digit pair.

Here is an example that is more relevant to corpus linguists. Suppose you want to simplify the annotation in `annotated.v`. Instead of XML annotation, you want a simpler scheme involving the POS tag appended to the word with an underscore. The pattern you want to match is defined as a generic tag:

```
"<w c5=\"(\\w+)\" hw=\"\\w+\" pos=\"\\w+\">(\\w+) ?</w>"
```

where `\\w+` ("one or more characters") stands for the specific values of the tag's attributes (`c5`, `hw`, and `pos`). The quotes must be escaped to be recognized as characters, not as character string boundaries. An optional space is added after the last occurrence of `\\w+` (that is where the word token would be) thanks to the quantifying expression `" ?"`. This guarantees that you will match all word tokens, including those that are not followed by a space but by punctuation marks. The elements that we are interested in are the `c5` tag and the word token, which is right between the starting tag and the end tag. Both elements should be placed between brackets for backreferencing. Because the `c5` tag is the first bracketed element, it is assigned number `1`, and because the word token is the second bracketed element, it is assigned number `2`. In the replacement argument, the word token (`\\2`) is followed by an underscore and the `c5` tag (`\\1`).

```
> annotated.v <- "<w c5=\"DT0\" hw=\"this\" pos=\"ADJ\">this </w><w c5=\"VBZ\" hw=\"be\"
pos=\"VERB\">is </w><w c5=\"AT0\" hw=\"an\" pos=\"ART\">an </w><w c5=\"AJ0\"
hw=\"annotated\" pos=\"ADJ\">annotated </w><w c5=\"NN1\" hw=\"sentence\"
pos=\"SUBST\">sentence</w><c c5=\"PUN\">."
> annotated.v2 <- gsub("<w c5=\"(\\w+)\" hw=\"\\w+\" pos=\"\\w+\">(\\w+) ?</w>", "\\2_\\1",
  annotated.v)
> annotated.v2
[1] "this_DT0 is_VBZ an_AT0 annotated_AJ0 sentence_NN1 <c c5=\"PUN\">."
> gsub("<c c5=\"(PUN)\">(\\.)", "\\2_\\1", annotated.v2)
[1] "this_DT0 is_VBZ an_AT0 annotated_AJ0 sentence_NN1 ._PUN"
```

Note that the punctuation tag was not captured by the regular expression. Again, backreferencing comes in handy.

4.4.9 Exact Matching with `strapply()`

The matching function `grep()` returns the vector element containing the match. Generally, when you load a corpus file, the separator is set to a newline (`sep="\n"`). When there is one corpus sentence per line, `grep()` returns not just the exact match but the whole sentence containing the match. This may not be what you always need.

To do exact matching, `strapply()` is the function you need. It is part of the `gsubfn` package, which you must obtain from CRAN and load beforehand (via `library(gsubfn)`). The simplified argument structure of `strapply()` is the following:

```
> strapply(X, pattern, backref, ignore.case=FALSE, perl = FALSE)
```

The `X` argument is the character vector where the matches are sought. The `pattern` argument is the character string you want to match. The `ignore.case` argument specifies whether the pattern matching is case sensitive or not. As you can see, it is case sensitive by default. The `perl` argument specifies whether the R engine should be used with perl-compatible regular expressions. By default, `perl` is set to `FALSE`, so do not forget to switch it to `TRUE` if you are using perl regular expressions. The function has a `backref` argument for backreference. Even though you already know from Sect. 4.4.8 what it can do, the way it is implemented in `strapply()` is slightly more complex and requires further explaining by means of a working example.

Your `CLSR` folder contains the BNC Baby. After loading the first corpus file (`A1E.xml`), we want to extract all the verbs from this file. This is easy as pie because the corpus is annotated with XML (see Sect. 4.4.8). Here is what a verb looks like in XML:

```
<w c5="VVZ" hw="reveal" pos="VERB">reveals </w>
```

All we need to do is (a) tell R to match all the tags that contain the string `pos="VERB"`, and (b) keep the actual verb in memory. With `strapply()`, these two steps can be done in one go. First, we load the `gsubfn` package.

```
> rm(list=ls(all=TRUE))
> library(gsubfn)

Loading required package:  proto
```

Next, we load the corpus file into the vector `A1E.xml`.

```
> A1E.xml <- scan(file="C:/CLSR/BNC_baby/A1E.xml", what="char", sep="\n") # Windows
> A1E.xml <- scan(file="/CLSR/BNC_baby/A1E.xml", what="char", sep="\n") # Mac
```

Then, we ask R to match `pos="VERB"` followed by a closing angle bracket (`>`), one or more word characters (`\\w+`) and a word boundary (`\\b`). The POS tag is used only to retrieve the verb. It is not needed in the returned match. To tell R to keep only the verb token, we must tag it as a backreference by enclosing it between brackets and call the match using the backreference argument (`backref`) of the `strapply()` function.

```
> verbs <- unlist(strapply(A1E.xml, "pos=\"VERB\">(\\w+)\\b", perl=TRUE, backref=-1))
> head(verbs, 30) # display the first thirty verbs
 [1] "reveals"      "is"           "leading"
 [4] "SEEMS"        "appear"       "has"
 [7] "made"         "has"          "investigated"
[10] "is"           "being"        "mounted"
[13] "says"         "is"           "had"
[16] "taken"        "would"        "have"
[19] "found"        "has"          "are"
[22] "has"          "investigated" "would"
[25] "keep"         "was"          "launched"
[28] "faced"        "regards"      "is"
```

The `backref` argument of `strapply()` is set to `-1`. This means that R remembers the string enclosed within the first set of brackets. Understanding how `backref` works is no easy task, all the more so as it is subject to change. You should therefore always check the `gsubfn` manual written by Gabor Grothendieck for changes.[4] If `backref` is set to zero, the entire match is returned. If `backref` is set to a positive integer *n*, the match and the first *n* backreferences are returned. If `backref` is set to a negative integer *-n*, only the first *n* backreferences are returned.

To obtain the number of verbs in the corpus file, use `length()`.

```
> length(verbs)
[1] 1497
```

The corpus file contains 1497 verbs.

Although `strapply()` does its job quite well, it is particularly slow when the character strings are long, even more so if it is used multiple times in a script or inside a loop. Chances are that if you work on a large corpus (such as the full BNC), you script will consume a lot of memory and take hours to run. Fortunately, Gabor Grothendieck has written a dramatically faster version of `strapply()`: `strapplyc()`.

4.4.10 Lookaround

If you are comfortable enough with the techniques seen above, it is likely that you will not be using lookaround very often. Lookaround comes in two flavors: lookahead and lookbehind. Lookahead constrains the match regarding what immediately follows or what does not immediately follow, whereas lookbehind constrains the match regarding what immediately precedes or does not immediately precede. Each subdivides into a positive and a negative version.

Suppose you have the vector `words.with.q`.

```
> words.with.q <- c("Iraq", "Iraq.", "Qatar", "Aqaba", "quest", "query", "aquatic")
```

You want to match both instances of “Iraq”. You cannot simply use `grep()` and look for `"q$"`, hoping the regular expression will match words that end in “q” because in the second instance, the character is not final but followed by a period.

```
> grep("q$", words.with.q, ignore.case=TRUE, value=TRUE)
[1] "Iraq"
```

[4] https://cran.r-project.org/web/packages/gsubfn/gsubfn.pdf.

We could try using positive lookahead to ask R to match a word that contains a "q" followed by a non-word character. The construct for positive lookahead is a pair of parentheses, with the opening parenthesis followed by a question mark, and an equals sign: `(?=text)`. For the assertion to be recognized, make sure that you specify `perl=TRUE`.

```
> grep("q(?=\\W)", words.with.q, ignore.case=TRUE, perl=TRUE, value=TRUE)
[1] "Iraq."
```

Unfortunately, only one match is returned: "Iraq.". This is because "Iraq" is not followed by a character or anything visible. The good news is that R sees a word boundary before and after each character string (see Sect. 4.4.6). What you can do is use positive lookahead to ask R to match a word that contains a "q" if the character is followed by a word boundary (`\\b`), i.e. in word-final position.

```
> grep("q(?=\\b)", words.with.q, ignore.case=TRUE, perl=TRUE, value=TRUE)
[1] "Iraq"  "Iraq."
```

At this stage, it is important to remember that lookaround expressions are zero-length assertions. In the above line of code, `\\b` is not consumed. Technically, you are not looking for "a q followed by a word boundary" but for "a q as long a it is followed by a word boundary". If you replace "q" with its uppercase equivalent, it becomes clear that the word boundary is not replaced, because it is not consumed in the match.

```
> gsub("q(?=\\b)", "Q", words.with.q, ignore.case=TRUE, perl=TRUE)
[1] "IraQ"    "IraQ."   "Qatar"   "Aqaba"
[5] "quest"   "query"   "aquatic"
```

Now suppose you want to match all words that contain an Arabic "q" in one go. You cannot simply use `grep()` and look for `"q[^u]"`, hoping the regular expression will match "Iraq", "Qatar", and "Aqaba" just because `[^u]` excludes the character "u".

```
> grep("q[^u]", words.with.q, ignore.case=TRUE, value=TRUE)
[1] "Iraq." "Qatar" "Aqaba"
```

Here, "Iraq.", "Qatar", and "Aqaba" are matched, but not "Iraq". This is because in the pattern `q[^u]`, `[^u]` has a length and R expects a character after "q". This does not apply to "Iraq", which has no character after "q". One easy option is to use negative lookahead, which matches a string if it is *not* followed by something else. We ask R to match the character "q" only if it is not followed by the character "u". The construct for positive lookbehing is a pair of parentheses, with the opening parenthesis followed by a question mark and an exclamation mark: `(?!text)`.

```
> grep("q(?!u)", words.with.q, ignore.case=TRUE, perl=TRUE, value=TRUE)
[1] "Iraq"  "Iraq." "Qatar" "Aqaba"
```

Finally, all matches are returned.

Lookbehind is not as flexible and not used as often as lookahead. It also suffers from serious limitations, as you will see below. In my experience, there is almost always an alternative to lookbehind. Consider the vector `prefix`.

```
> prefix <- c("incomplete", "complete", "illogical", "logical",
+             "irresponsible", "responsible","impossible", "possible")
```

You want to extract only those adjectives that have a prefix meaning 'not' or 'the converse of'. This can be done with positive lookbehind: look for a word (`\\w+\\b`) if a word boundary followed by the character "i" and one word character (`\\bi\\w`) immediately precede it.

```
> grep("(?<=\\bi\\w)\\w+\\b", prefix, perl=TRUE, value=TRUE)
[1] "incomplete"    "illogical"
[3] "irresponsible" "impossible"
```

You obtain the same result more elegantly without positive lookbehind.

```
> grep("\\bi\\w+\\b", prefix, perl=TRUE, value=TRUE)
[1] "incomplete"    "illogical"
[3] "irresponsible" "impossible"
```

Positive lookbehind is faster and perhaps more useful with `gsub()` if you want to isolate the prefixes.

```
> gsub("(?<=\\bi\\w)\\w+\\b", "-", prefix, perl=TRUE) # with positive lookbehind
[1] "in-"         "complete"    "il-"
[4] "logical"     "ir-"         "responsible"
[7] "im-"         "possible"
> gsub("(\\bi\\w)\\w+\\b", "\\1-", prefix, perl=TRUE) # without positive lookbehind
[1] "in-"         "complete"    "il-"
[4] "logical"     "ir-"         "responsible"
[7] "im-"         "possible"
```

Suppose you have the vector `plural.singular`, which contains the singular and plural forms of a few words and an oddity: the genitive form "John's". The construct for positive lookbehind is a pair of parentheses, with the opening parenthesis followed by a question mark, the "less than" symbol, and an equals sign: `(?<=text)`.

```
> plural.singular <- c("car", "cars", "book", "books", "cap", "caps", "John's")
```

You want to match singular nouns. "John" is one of them, but you do not want to match "'s". One option involves looking for a word (`\\b\\w+\\b`) and adding a constraint before the closing word boundary: `[^s]` (not the character "s"). Another option involves negative lookbehind, which matches a string is it is *not* preceded by something else. The construct for negative lookbehind is `(?<!text)`: a pair of parentheses, with the opening parenthesis followed by a question mark, the "less than" symbol, and an exclamation mark. With lookaround, we can match a word if there is no "s" immediately preceding the closing word boundary: `\\b\\w+(?<!s)\\b`. Apparently, both options yield the same output.

```
> # without lookaround
> grep("\\b\\w+[^s]\\b", plural.singular, ignore.case=TRUE, perl=TRUE, value=TRUE)
[1] "car"    "book"   "cap"    "John's"
> # with lookaround
> grep("\\b\\w+(?<!s)\\b", plural.singular, ignore.case=TRUE, perl=TRUE, value=TRUE)
[1] "car"    "book"   "cap"    "John's"
```

Yet, remember that lookaround assertions have no length. In other words, what is between the parentheses is not returned. The difference between the regular expressions with and without negative lookbehind becomes obvious when you replace the singular nouns with, say, one asterisk.

```
> # without lookaround
> gsub("\\b\\w+[^s]\\b", "*", plural.singular, ignore.case=TRUE, perl=TRUE)
[1] "*"     "cars"  "*"     "books" "*"
[6] "caps"  "*s"
> # with lookaround
> gsub("(\\b\\w+(?<!s)\\b)", "*", plural.singular, ignore.case=TRUE, perl=TRUE)
[1] "*"     "cars"  "*"     "books" "*"
[6] "caps"  "*'s"
```

Without lookaround, the match is "John"', with the apostrophe. With lookaround, the match is what you really want: "John", without the apostrophe.

A known issue with lookbehind is that the assertion must be "fixed length". In the vector `birds` below, we want to match only those birds that have a prefix, e.g. *black* in *blackbird*, as opposed to the adjective *black* in *black bird*.

```
> birds <- c("black bird", "blackbird", "hummingbird", "cowbird",
+            "mousebird", "king bird-of-paradise")
> grep("\\b(?<=\\w+)bird\\b", birds, perl=TRUE, value=TRUE)

Warning in grep("\\b(?<=\\w+)bird\\b", birds, perl = TRUE, value = TRUE): PCRE pattern compilation
error
'lookbehind assertion is not fixed length'
at ')bird\b'
```

R issues a warning because in the positive lookbehind assertion `"\\b(?<=\\w+)bird\\b"`, `"\\w+"` matches character strings of various lengths.

```
> nchar(c("black", "humming", "cow", "mouse"))
[1] 5 7 3 5
```

Because of that serious limitation, lookbehind is seldom convenient for the kind of character string processing involved in corpus linguistics.

Tab. 4.5 summarizes the lookaround assertions and how each assertion is prefixed and used.

Table 4.5: Summary table for lookaround assertions

lookaround assertion	prefix	usage
positive lookahead	`?=`	`(?=text)`
negative lookahead	`?!`	`(?!text)`
positive lookbehind	`?<=`	`(?<=text)`
negative lookbehind	`?<!`	`(?<!text)`

Exercises

4.1. Exact matching

Consider the following vector:

```
> catmat <- c("the", "cat", "sat", "on", "the", "mat")
```

Find two ways of matching `cat`, `mat`, but not `sat`.

4.2. Word count

The text file `gnu.txt`, available in `CLSR/chap4`, is an excerpt from the preamble to the GNU General Public License (https://www.gnu.org/licenses/gpl-3.0.en.html). Count the number of times that the definite article *the* occurs in the text.

4.3. Exact matching with an annotated file

Load the corpus file `102CTL002_annotated.txt` from `CLSR/chap4` into a vector named `corp.file`. The corpus file is a letter from the Open American National Corpus. The text is annotated for part of speech according to CLAWS7. In this exercise, you have to retrieve verb forms thanks to the POS tags using regular expressions. The tags should not be part of the output. Don't forget to inspect the CLAWS7 tagset before writing the code. It is available from `http://ucrel.lancs.ac.uk/claws7tags.html`

(a) retrieve the past participle forms of all lexical verbs;
(b) retrieve all the verb forms of *be*;
(c) retrieve the 3SG forms of *be*, *do*, and *have* in the simple present.

The tags should not be retrieved.

Chapter 5
Applied Character String Processing

Abstract In this chapter, you will learn how to handle text material by combining the R techniques that you have learned in the previous chapter.

5.1 Introduction

You are now all set for digging corpora and extracting linguistic data in textual form. The sections below walk you through basic corpus linguistics operations. All of them involve the functions described above.

5.2 Concordances

A concordance table displays a word or a pattern of words in context. The context is a window of words to the left of the pattern (known as the preceding context or left context) and another window of words (known as the following context). In corpus linguistics lingo, a concordance is also known as a KWIC display, where the acronym stands for 'Key Word(s) In Context'.

A classic example involves a comparison of *-ic* and *-ical* adjectives (Gries 2001). Tab. 5.1 is a sample concordance of *alphabetic* and *alphabetical* in the British National Corpus. A cursory look at the concordance confirms that *alphabetic* refers to the use of letters of the alphabet, whereas *alphabetical* relates to the use of an alphabet as an ordering system.

5.2.1 A Concordance Based on an Unannotated Corpus

First, let us make a concordance of words based on *blood* in Bram Stoker's *Dracula*: *blood*, *bloody*, *blood-ied*, *bleed*, *bleeding*, etc. The text file can be downloaded from Project Gutenberg: `https://www.gutenberg.org/files/45839/45839-0.txt`. Save the text file in the `CLSR/chap5` folder. Name this file `dracula.txt`. If you encounter an encoding issue when you open the file in your text editor, please refer to Sect. 3.3.2.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_5

Table 5.1: A sample concordance of *alphabetic* and *alphabetical* in the BNC

#	left context	word	right context
1	specify the equation press the end key which is between the	alphabetic	and the numeric key pads , that will then submit that request
2	an integer between 0 and 9 , and A is an	alphabetic	character . # NISS No issue has been specified for the referenced
3	particular the notion that a word is a contiguous sequence of	alphabetic	characters . Amsler shows that this notion ignores important
4	... It was only in the days of the first widespread	alphabetic	culture that the idea of ' logic ' appears to have arisen
5	records than for signs . In most languages with writing systems	alphabetic	fingerspelling has been available for over two hundred years .
6	of icebreakers when she noticed that the control panel had an	alphabetic	keyboard. Can't hurt to try , she thought , unjacking
7	for the widespread development of the particular form of	alphabetic	literacy evident in Greece must clearly be sought in the social
8	Retrieval from a conventional filing system relies on either	alphabetic	or numeric indexation but the use of a computer enables data to
9	on . Fingerspelling as it exists today consists of a direct	alphabetic	representation of the language as it would be written down . In
10	describe . According to Goody and Watt , the development of	alphabetic	script and its wide diffusion throughout society – the two
11	individuals or items which can be symbolized by a number or	alphabetical	abbreviation . For example , 929 Schil Biography of Schiller 820
12	Filofax – but you probably do need a college diary or	alphabetical	address book with space for telephone numbers , etc . Finally ,
13	A to Z OF FISH HEALTH # JERZY GAWOR continues his	alphabetical	advice on fish health problems . # Top : Tablet food forms
14	this context . Abish 's use of arbitrary formal limits in	Alphabetical	Africa (1974) ' becomes the vehicle for an adventurous plot
15	index to post code directories. # 5. # Telephone directories –	alphabetical	and classified . # 6. # Telephone Dialling Codes – an essential
16	in order to arrange it . A classified rather than an	alphabetical	approach was necessary in the index because an internationally
17	jazz titles from early this and late last year , in	alphabetical	artist order . Old faces mix with the new but all have
18	when I sell it again I can put the type in	alphabetical	ascending order (SP:PS0H9) Mm (SP:PS0H9) so it 'll be
19	. These six should ideally be selected at random from an	alphabetical	class list . The observer must ensure that the six named in
20	brother , Master Richard of Stainby , author of a revised	alphabetical	concordance to the Bible , and the great scholar , Alexander of

5.2.1.1 First Try

First, load the text file into a character vector, and inspect.

```
> rm(list=ls(all=TRUE)) # clear memory
> dracula <- scan(file="C:/CLSR/chap5/dracula.txt", what="char", sep="\n") # Windows
> dracula <- scan(file="/CLSR/chap5/dracula.txt", what="char", sep="\n") # Mac
> head(dracula)
```

Electronic texts from Project Gutenberg come with a header and a footer, a table of contents and editorial information, which we do not want. We have to tell R to ignore those by pointing at the relevant interval in the file where the novel is located. We could open the file in a text editor and manually delete the unwanted text. However, for expository reasons, let me show you how to do it in R.

When you loaded the text file, you specified the newline as a separator. This means that each line is indexed as a vector element. We can use the index to tell R that the novel is located between the first novel line and the last. The novel proper starts at "CHAPTER I." and ends at "/Jonathan Harker./". We retrieve the line numbers with `grep()` and tell R that the novel spans between these two landmarks.

```
> start <- grep("^CHAPTER I.$", dracula) ; start
[1] 132
> end <- grep("^\\/Jonathan Harker.\\/$", dracula) ; end
[1] 13014
> novel <- dracula[start:end] #join the beginning and the end of the novel
> head(novel) # inspect
[1] "CHAPTER I."
[2] "/Jonathan Harker's Journal./"
[3] "(Kept in shorthand.)"
[4] "3 May. Bistritz.--Left Munich at 8.35 p.m. on 1st May, arriving at"
[5] "Vienna early next morning; should have arrived at 6.46, but train was"
[6] "an hour late. Buda-Pesth seems a wonderful place, from the glimpse"
> tail(novel) # inspect
[1] "said, with our boy on his knee:--"
```

```
[2] "We want no proofs; we ask none to believe us! This boy will some day"
[3] "know what a brave and gallant woman his mother is. Already he knows her"
[4] "sweetness and loving care; later on he will understand how some men so"
[5] "loved her, that they did dare much for her sake."
[6] "/Jonathan Harker./"
```

One aspect of the text file that corpus linguists dislike is that the line breaks do not correspond to anything linguistically tangible (such as a clause or a sentence for instance). They correspond to the line breaks of the original 1897 edition of *Dracula*. To get things organized, the next step is to convert the vector `novel`, which contains as many elements as there are line breaks into a character vector that contains one element (or, in R parlance, a character vector of length 1). With `paste()` and the `collapse` argument, it is a piece of cake.

```
> novel.vector <- paste(novel, collapse=" ")
```

Normally, the next step is to convert the text into individual words by splitting at non-word characters: `unlist(strsplit(novel.vector, "\\W+"))`. This process is known as tokenization. Yet, we should not rush to this step because it removes all punctuation, and we need some of it. Indeed, some *blood*-based lexemes may be compound words: e.g. *blood-curdling*, *cold-blooded*. See what happens when we split the toy vector `blood.sentence` at non-word characters and then look for all *blood*-based lexemes, including compounds.

```
> blood.sentence <- "a sentence with the words blood, blood-curdling, cold-blooded)"
> blood.words <- unlist(strsplit(blood.sentence, "\\W+"))
> grep("blood", blood.words, value=TRUE)
[1] "blood"   "blood"   "blooded"
> grep("(\\w+)?-?blood-?(\\w+)?", blood.words, value=TRUE)
[1] "blood"   "blood"   "blooded"
```

Only simple lexemes are returned. If you inspect the vector `blood.words`, you will see that all hyphens have been consumed by `strsplit()`. Therefore, they are not available anymore, and compounds have been split into distinct lexemes.

There are two kinds of hyphens in the text file, as transcribed by the Project Gutenberg volunteers: double hyphens and single hyphens. The double hyphens stand for an en dash or em dash.[1] Double hyphens can be replaced by a space. Single hyphens can be deleted for the purpose of the concordance (you can put them back later).

```
> # remove hyphens
> novel.vector <- gsub("--", " ", novel.vector)
> novel.vector <- gsub("-", "", novel.vector)
```

You may now split the text at non-word characters.

```
> # split at non word characters
> novel.vector.words <- unlist(strsplit(novel.vector, "\\W+"))
> head(novel.vector.words) # inspect
[1] "CHAPTER"  "I"        "Jonathan" "Harker"
[5] "s"        "Journal"
> tail(novel.vector.words) # inspect
[1] "much"     "for"      "her"      "sake"
[5] "Jonathan" "Harker"
```

It is now time to look for the desired character strings with `grep()`. Rather than use the function to extract the matches by specifying `value=TRUE`, we will set the argument to `FALSE` so as to retrieve the

[1] See http://www.thepunctuationguide.com/en-dash.html.

positions of the matches. Since we do not have a clear idea of what words we will find, we had better make a wide search. The pattern

\\b\\w*bl(oo|ee?)d\\w*\\b

is broad enough to capture a wide variety of *blood*-based expressions like *blood*, *bloodless*, *cold-blooded*, *bloody*, *bleed*, *bleeding*, or *bled*. Make sure you delimit the match with word boundaries (\\b) and not non-word characters (\\W), otherwise you will miss matches that are at the beginning or the end of a character string. The part bl(oo|ee?)d is designed to match *blood*, *bleed*, or *bled* (the question mark in ee? makes the second "e" optional). The prefix \\b\\w* and the suffix \\w*\\b match these lexemes when they are parts of compounds.

In general, it is best to test an expression on a toy vector that includes potential matches and mismatches.

```
> toy.vector.1 <- c("blood", "coldblooded", "bloodless", "bleed", "bleeding", "bled", "mumbled")
> grep("\\b\\w*bl(oo|ee?)d\\w*\\b", toy.vector.1,  value=TRUE)
[1] "blood"       "coldblooded" "bloodless"
[4] "bleed"       "bleeding"    "bled"
[7] "mumbled"
```

Unfortunately, the regular expression is too broad: it matches *mumbled* (which also contains the string *bled*) as it would *grumbled* or *fumbled*. This is because *mum-* is matched by \\b\\w* in

\\b\\w*bl(oo|ee?)d\\w*\\b.

The solution is to use line anchors so that nothing precedes or follows *bled*: ^bled$. Rather than complexify the pattern further, it is best to run two separate queries and concatenate them afterwards.

```
> # retrieve word positions
> node.positions.1 <- grep("\\b\\w*bl(oo|ee)d\\w*\\b", novel.vector.words) # first query
> node.positions.2 <- grep("^bled$", novel.vector.words) # second query
> node.positions <- c(node.positions.1, node.positions.2) # concatenate both queries
> head(node.positions) # inspect
[1]  8954 10851 12075 12172 12203 12343
```

We are now going to ask R to iterate over the text file to match the positions with their corresponding nodes by means of a for loop (see Sect. 2.10). More precisely, the loop will tell R to:

- access each match;
- get the left context of each match;
- get the right context of each match;
- print the results into a text file.

Beforehand, prepare an empty text file named dracula_conc.txt in the chap5 subfolder.

```
> output.conc <- "C:/CLSR/chap5/dracula_conc.txt" # Windows
> output.conc <- "/CLSR/chap5/dracula_conc.txt" # Mac
```

Prepare a tab-delimited header with cat(). Store the path to the file in a vector (output.conc).

```
> cat("LEFT CONTEXT\tNODE\tRIGHT CONTEXT\n", file=output.conc) # tab-delimited header
```

Create a vector where you specify a value (in word numbers) for the left and right contextual windows.

```
> context <- 10 # specify a window of ten words before and after the match
```

At this stage, it is essential that you understand how this value will be used. Each match is assigned a position in the form of a numeric value (all positions are contained in `node.position`). By setting context to 10, you want to retrieve a window of ten words to the left and to the right of each match.

Finally, enter the loop.

```
> for (i in 1:length(node.positions)){ # access each match...
+   # access the current match
+  node <- novel.vector.words[node.positions[i]]
+  # access the left context of the current match
+  left.context <- novel.vector.words[(node.positions[i]-context):(node.positions[i]-1)]
+  # access the right context of the current match
+  right.context <- novel.vector.words[(node.positions[i]+1):(node.positions[i]+context)]
+  # concatenate and print the results
+  cat(left.context,"\t", node, "\t", right.context, "\n", file=output.conc, append=TRUE)
+ }
```

When you open `dracula_conc.txt` with your spreadsheet software, you will see that the script seems to have worked fine in this case.

5.2.1.2 Potential Issues

Before congratulating yourself, you need to tackle two potential issues if you are to run the script on a different text file. The first issue arises if the node is among the very first/last words of the text file and the contextual window exceeds the boundaries of the text file. Let us use another toy vector to illustrate this and see how we can handle it.

```
> rm(list=ls(all=TRUE))
> toy.vector.2 <- c("The blood of snakes is toxic. Vampires drink the blood of animals.")
```

In the first sentence, *blood* is in second position. If we ask R to go back three words before it, this is bound to be a problem. Indeed, if we start from position 2 and go back three positions, we obtain a negative position (-1). In the second sentence, *blood* is in tenth position. The whole vector has only twelve word positions. If we ask R to go forth three words after the tenth position, it will not be able to find the 13th word position of the vector because there is none. In its current state, our script does not take this situation into account. If you run it, R issues an error message.

```
> split.tv2 <- unlist(strsplit(toy.vector.2, "\\W+"))
> node.positions <- grep("blood", split.tv2, ignore.case=TRUE)
> context <- 3
> for (i in 1:length(node.positions)){ # access each match...
+   # access the current match
+  node <- split[node.positions[i]]
+  # access the left context of the current match
+  left.context <- split[(node.positions[i]-context):(node.positions[i]-1)]
+  # access the right context of the current match
+  right.context <- split[(node.positions[i]+1):(node.positions[i]+context)]
+  # print the results
+  cat(left.context,"\t", node, "\t", right.context, "\n", append=TRUE)
+ }
> # you're in for an error message...
```

The second issue is a specific instance of the first issue. It arises if the match is the first or the last word of the text file.

```
> rm(list=ls(all=TRUE))
> toy.vector.3 <- c("Blood is the red liquid that flows through the bodies of people and animals.
+                   Vampires drink blood.")
```

In the first sentence of this vector, *blood* does not have a left context. The same word does not have a right context in the second sentence. The above script will output something strange because it will ask R to provide a left/right context for the node, but that context does not exist.

```
> split.tv3 <- unlist(strsplit(toy.vector.3, "\\W+"))
> node.positions <- grep("blood", split.tv3, ignore.case=TRUE)
> context <- 3
> for (i in 1:length(node.positions)){ # access each match...
+   # access the current match
+  node <- split.tv3[node.positions[i]]
+  # access the left context of the current match
+  left.context <- split.tv3[(node.positions[i]-context):(node.positions[i]-1)]
+  # access the right context of the current match
+  right.context <- split.tv3[(node.positions[i]+1):(node.positions[i]+context)]
+  # print the results
+  cat(left.context,"\t", node, "\t", right.context, "\n", append=TRUE)
+ }
the red liquid that flows through the bodies of people and animals Vampires drink blood Blood is
the red animals Vampires drink blood NA NA NA
```

5.2.1.3 Addressing Issues

To tackle these issues, we have to tell R what to do if such situations arise. Each issue will therefore be accounted for by means of an `if` statement (see Sect. 2.11).

Let us address the first issue. The `if...else` statement below prevents R from going too far back to retrieve context if the difference between the position of the node and the context is negative. The context is reset to a sequence between the first position available in the vector and the position right before the node. If the condition is not true, R proceeds normally.

```
> #if (node.positions[i] < context) left.context <- split.tv2[1:(node.positions[i]-1)]
> #else left.context <- split.tv2[(node.positions[i]-context):(node.positions[i]-1)]
```

A second `if...else` statement prevents R from going too far ahead to retrieve context if the sum of the position of the node and the context exceeds the length of the vector. The context is reset to a sequence between the position right after the node and the last position available in the vector.

```
> #if ((node.positions[i] + context) > length(split.tv2))
> #  right.context <- split.tv2[(node.positions[i]+1):length(split.tv2)]
> #else right.context <- split.tv2[(node.positions[i]+1):(node.positions[i]+context)]
```

Let us check the validity of the script with `toy.vector.2`.

```
> rm(list=ls(all=TRUE))
>
> toy.vector.2 <- c("The blood of snakes is toxic. Vampires drink the blood of animals.")
> split.tv2 <- unlist(strsplit(toy.vector.2, "\\W+"))
> node.positions <- grep("blood", split.tv2, ignore.case=TRUE)
>
> context <- 3
>
> for (i in 1:length(node.positions)){ # access each match...
+
+   # access the current match
+   node <- split.tv2[node.positions[i]]
+
+   # access the left context of the current match
+   if (node.positions[i] < context) left.context <- split.tv2[1:(node.positions[i]-1)]
+   else left.context <- split.tv2[(node.positions[i]-context):(node.positions[i]-1)]
```

```
+
+   # access the right context of the current match
+   if ((node.positions[i] + context) > length(split.tv2))
+     right.context <- split.tv2[(node.positions[i]+1):length(split.tv2)]
+   else right.context <- split.tv2[(node.positions[i]+1):(node.positions[i]+context)]
+
+   # print the results
+   cat(left.context,"\t", node, "\t", right.context, "\n", append=TRUE)
+ }
The   blood   of snakes is
Vampires drink the   blood   of animals
```

Let us now address the second issue. The `if...else` statement below assigns a `NULL` value to `left.context` if the node is in first position. If that condition is not true, R proceeds normally.

```
> #if (node.positions[i]==1) left.context <- NULL
> #else left.context <- split.tv2[1:(node.positions[i]-1)]
```

We need another `if...else` statement to assign a `NULL` value to `right.context` if the node is in last position. If the condition is not true, R proceeds normally.

```
> #if (node.positions[i]==length(split.tv2)) right.context <- NULL
> #else right.context <- split[(node.positions[i]+1):length(split)]
```

Let us now check the validity of the script with `toy.vector.3`.

```
> rm(list=ls(all=TRUE))
>
> toy.vector.3 <- c("Blood is the red liquid that flows through the bodies of people and animals.
+                   Vampires drink blood.")
> split.tv3 <- unlist(strsplit(toy.vector.3, "\\W+"))
> node.positions <- grep("blood", split.tv3, ignore.case=TRUE)
>
> context <- 3
>
> for (i in 1:length(node.positions)){ # access each match...
+
+   # access the current match
+   node <- split.tv3[node.positions[i]]
+
+   # access the left context of the current match
+   if (node.positions[i]==1) left.context <- NULL
+   else left.context <- split.tv3[(node.positions[i]-context):(node.positions[i]-1)]
+
+   # access the right context of the current match
+   if (node.positions[i]==length(split.tv3)) right.context <- NULL
+   else right.context <- split.tv3[(node.positions[i]+1):(node.positions[i]+context)]
+
+   # print the results
+   cat(left.context,"\t", node, "\t", right.context, "\n", append=TRUE)
+ }
   Blood   is the red
animals Vampires drink   blood
```

Problem solved.

5.2.1.4 The Full Script

Because the second issue is a subcase of the first one, the `if...else` statement that addresses it will have to be nested in the `if...else` statement that addresses the first issue.

```
> rm(list=ls(all=TRUE)) # clear memory
>
> dracula <- scan(file="/CLSR/chap5/dracula.txt", what="char", sep="\n", fileEncoding="UTF-8")
>
> start <- grep("^CHAPTER I.$", dracula) ; start
> end <- grep("^\\/Jonathan Harker.\\/$", dracula) ; end
> novel <- dracula[start:end]
>
> novel.vector <- paste(novel, collapse=" ")
> novel.vector <- gsub("--", " ", novel.vector)
> novel.vector <- gsub("-", "", novel.vector)
>
> split <- unlist(strsplit(novel.vector, "\\W+"))
>
> node.positions.1 <- grep("\\b\\w*bl(oo|ee)d\\w*\\b", split) # first query
> node.positions.2 <- grep("^bled$", split) # second query
> node.positions <- c(node.positions.1, node.positions.2)
>
> output.conc <- "/CLSR/chap5/dracula_conc.txt" # Mac
> cat("LEFT CONTEXT\tNODE\tRIGHT CONTEXT\n", file=output.conc)
>
> context <- 10
>
> for (i in 1:length(node.positions)){ # access each match...
+
+   # access the current match
+   node <- split[node.positions[i]]
+
+   # access the left context of the current match
+   if (node.positions[i] < context)
+     if (node.positions[i]==1) left.context <- NULL
+     else left.context <- split[1:(node.positions[i]-1)]
+   else left.context <- split[(node.positions[i]-context):(node.positions[i]-1)]
+
+   # access the right context of the current match
+   if ((node.positions[i] + context) >= length(split))
+     if (node.positions[i]==length(split)) right.context <- NULL
+     else right.context <- split[(node.positions[i]+1):length(split)]
+   else right.context <- split[(node.positions[i]+1):(node.positions[i]+context)]
+
+   # print the results
+   cat(left.context,"\t", node, "\t", right.context, "\n", file=output.conc, append=TRUE)
+ }
```

Fig. 5.1 is what the file looks like when you open it with a spreadsheet software. *Blood* is used figuratively more than it is used literally. Someone doing a literary study of the status of blood in *Dracula* will find a concordance useful to compare the distribution of *blood*-based words across the whole novel.

	A	B	C
1	LEFT CONTEXT	NODE	RIGHT CONTEXT
2	all this region that has not been enriched by the	blood	of men patriots or invaders In old days there were
3	saw that the cut had bled a little and the	blood	was trickling over my chin I laid down the razor
4	right to be proud for in our veins flows the	blood	of many brave races who fought as the lion fights
5	the dying peoples held that in their veins ran the	blood	of those old witches who expelled from Scythia had mated
6	or what witch was ever so great as Attila whose	blood	is in these veins He held up his arms Is
7	more gladly than we throughout the Four Nations received the	bloody	sword or at its warlike call flocked quicker to the
8	and again though he had to come alone from the	bloody	field where his troops were being slaughtered since he knew
9	we threw off the Hungarian yoke we of the Dracula	blood	were amongst their leaders for our spirit would not brook
10	sir the Szekelys and the Dracula as their heart s	blood	their brains and their swords can boast a record that
11	underlying the sweet a bitter offensiveness as one smells in	blood	I was afraid to raise my eyelids but looked out

Fig. 5.1: A snapshot of a concordance of *blood*-based words in *Dracula*

5.2.2 A Concordance Based on an Annotated Corpus

We want to make a concordance of *almost* and *nearly* in the BNC Baby. The procedure is similar to what you need to do for the concordance of an unannotated corpus. I assume you have read Sect. 3 and know how to obtain the corpus. The BNC Baby consists of 182 XML text files. The texts are classified with respect to four genres: academic, fiction, newspaper and conversation. Each genre is a one-million-word sample. Each word is annotated for lemma information and POS-tags.

TEI-compliant corpora make it easier for corpus linguists to make concordances. Chances are that you will not have to worry about the kind of issues that we dealt with in the previous section because each sentence has its own line. Unless you want to specify a window of words before and after the match, as we did above, all you need to do is to look for the desired word or expression, and delimit your match with tab stops. Thus, if the match is in first or last position, it will not be a problem. The main difficulty occurs when there is more than one match per corpus line/sentence.

The procedure breaks down into the following steps:

- select the corpus files;
- prepare an empty character vector to store the results;
- access each corpus file with a `for` loop;
- isolate the corpus sentences from the rest of the corpus file;
- check if the sentences in the corpus file contain matches;
- if they do, extract each match and its context;
- extract information about the match (e.g. the name of the corpus file, the mode, etc.) and append that information to the match;
- customize the output;
- export the output into a tab-delimited text file.

5.2.2.1 The Naive Way

To load the corpus files faster, I have modified the native hierarchy of the directories of the BNC Baby. More specifically, I have moved the corpus files from the fours subdirectories `aca`, `dem`, `fic`, and `news` into a single directory named `BNC_baby`, which I have placed in the `CLSR` folder, as shown in Fig. 5.2.

In Sect. 4.2.2, you learned how to load several text files into R at the same time. Let us do it using the `list.files()` function, after clearing R's memory.

```
> rm(list=ls(all=TRUE))
>
> corpus.files <- list.files(path="C:/CLSR/BNC_baby", pattern="\\.xml$", full.names=TRUE) # Windows
> corpus.files <- list.files(path="/CLSR/BNC_baby", pattern="\\.xml$", full.names=TRUE) # Mac OS
```

The `pattern` argument contains the regular expression `"\\.xml$"` to make sure that no hidden files such as those that your operating system might store without your knowledge are retrieved. When you inspect `corpus.files`, you see that it contains the full path to all 182 corpus files.

```
> head(corpus.files)
[1] "/CLSR/BNC_baby/A1E.xml"
[2] "/CLSR/BNC_baby/A1F.xml"
[3] "/CLSR/BNC_baby/A1G.xml"
[4] "/CLSR/BNC_baby/A1H.xml"
[5] "/CLSR/BNC_baby/A1J.xml"
```

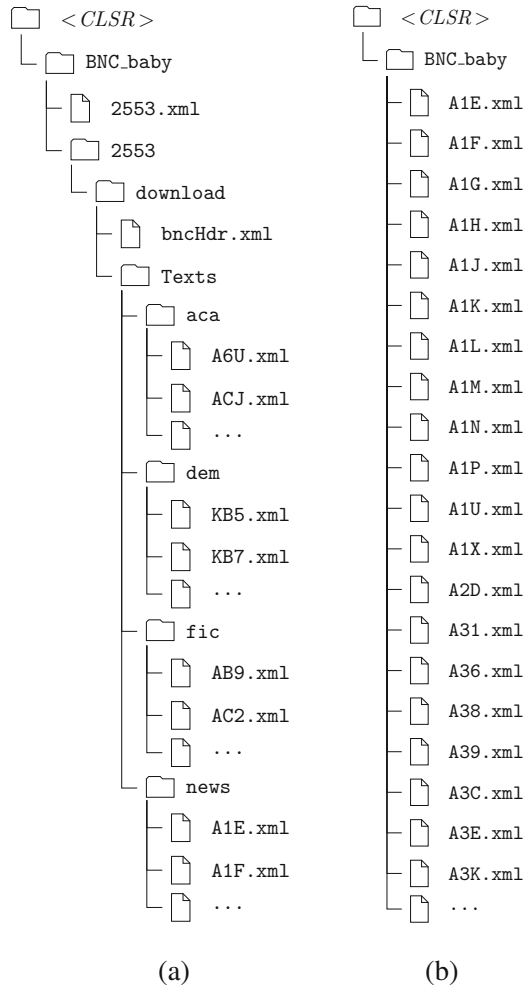


Fig. 5.2: Modifying the native hierarchy of the BNC Baby directories. (**a**) Original; (**b**) simplified

```
[6] "/CLSR/BNC_baby/A1K.xml"
> length(corpus.files)
[1] 182
```

Now we need to create an empty character vector to collect all the matches.

```
> all.matches <- character() # create an empty character vector
```

Thanks to a `for` loop, we will access each corpus file and collect all that we need.

Before we do, let us look briefly at the structure of a BNC Baby corpus file. Open `A1E.xml` with your text editor. The first line contains the TEI header, which is between the tags <teiHeader> and </teiHeader>, and specific information about the source and the structure of the corpus file. The second line begins with the start tag for a written text element (<wtext>), which bears a type attribute, the value of which is NEWS,

the code used for texts derived from newspapers. Next comes the elements, which has two attributes: level and n. The n attribute supplies an identifying number and a name used in the text for a given division (a page number in this case). The level attribute is an indication of what the text is part of, i.e. a chapter, a chapter part, a chapter subpart, etc. The <head> element contains the name of the element. Line 3 marks the beginning of the body of the corpus material that we are interested in. Originally, <s> stands for a segment. For convenience's sake, we can say that it also stands for a sentence. The n attribute assigns a number to the sentence to which it is attached. This means that sentences are indexed sequentially.

We want R to access each file in turn, i.e. from the first corpus file to the 182th corpus file. The `for` loop will therefore be introduced by the following code: `for (i in 1:length(corpus.files))`. The identifier will take on the values between 1 and 182.

Once inside the loop, R will load the current corpus file,

```
> current.corpus.file <- scan(corpus.files[i], what="char", sep="\n", quiet=TRUE)
```

and keep only the corpus sentences from this file.

```
> current.sentences <- grep("<s n=", current.corpus.file, value=TRUE)
```

The next step is to grab the matches with `grep()`. We are looking for two function words, *each* and *every* which, in XML/CLAWS5 format, look like this:

```
<w c5="DT0" hw="each" pos="ADJ">each </w>
<w c5="AT0" hw="every" pos="ART">every </w>
```

Of course, we could run two queries (one for *each* and one for *every*) and concatenate them afterwards but, thanks to regular expressions, and because the tags are similar, we just need to make one query for both forms.

```
<w c5=\"(DT0|AT0)\" hw=\"(each|very)\" pos=\"(ADJ|ART)\">(each|every) ?</w>
```

Note that all double quotes are escaped, so that R does not mix them up with character string delimiters. Here is the detail of what each part of the expression means:

- `(DT0|AT0)`: "DT0 or AT0";
- `(each|every)`: "*each* or *every*";
- `(ADJ|ART)`: "ADJ or ART";
- `(each|every) ?`: "*each* or *every* followed by one optional space".

The space after `(each|every)` is made optional because if the match is followed by a comma or a period, there will be no space between the last character of the match and the punctuation mark. The regular expression could be less specific, e.g.

```
<w c5=\"\\w{2}\\d" hw=\"(each|every)\" pos=\"\\w{3}\">(each|every) ?</w>
```

but this is risky as you do not know what exception the corpus has in store for you. Furthermore, the point of using an annotated corpus is precisely to use the annotation. I advise you to use the tags as much as you can. Once you are happy with your regular expression, store it in a vector so that it can be called easily from inside the loop.

```
> regex.match <- "<w c5=\"(DT0|AT0)\" hw=\"(each|every)\" pos=\"(ADJ|ART)\">(each|every) ?</w>"
```

The next line in the loop looks like this:

```
> current.matches <- grep(regex.match, current.sentences, ignore.case=TRUE, value=TRUE)
```

Next, insert a condition in case the query returns no match in the current corpus file:

```
> if (length(current.matches)==0) { next }
```

If there are matches, prefix the name of the corpus file to each match and store the results in a vector:

```
> current.matches.all <- paste(basename(corpus.files[i]), current.matches, sep="\t")
```

Finally, collect all the matches in the empty character vector that you prepared beforehand and exit the loop.

```
> all.matches <- c(all.matches, current.matches.all)
```

Here is what the full loop looks like.

```
>  for (i in 1:length(corpus.files)) { # enter the for loop
+
+  # load the current corpus file
+    current.corpus.file <- scan(corpus.files[i], what="char", sep="\n", quiet=TRUE)
+
+  # keep only the corpus sentences from the current file
+    current.sentences <- grep("<s n=", current.corpus.file, value=TRUE)
+
+  # retrieve the matches from the current file
+    current.matches <- grep(regex.match, current.sentences, ignore.case=TRUE, value=TRUE)
+
+  # check whether there are any matches
+    if (length(current.matches)==0) { next }
+
+  # if yes, prefix the name of the corpus file to each match and store the results in a vector
+    current.matches.all <- paste(basename(corpus.files[i]), current.matches, sep="\t")
+
+  # collect all the matches in the character vector
+    all.matches <- c(all.matches, current.matches.all)
+
+  } # exit the for loop
```

At this stage, all the matches are stored in `all.matches`. They are in raw format, which we need to process. First, we insert tab stops before and after each match with `gsub()` and backreferencing.

```
> all.matches.tab <- gsub(regex.match, "\t\\4\t", all.matches, ignore.case=TRUE)
```

Second, we insert a tab after the sentence number, so that each sentence number has its own column.

```
> all.matches.tab.2 <- gsub("<s n=\"(\\d+)\">", "sentence \\1\t", all.matches.tab,
+                           ignore.case=TRUE)
```

Third, we remove all tags with lazy matching (Sect. 4.4.7).

```
> all.matches.tab.3 <- gsub("<.*?>", "", all.matches.tab.2, ignore.case=TRUE)
```

Finally, we remove unwanted spaces before and after the tabulations.

```
> all.matches.tab.4<-gsub(" *(\t) *", "\\1", all.matches.tab.3)
```

All that is left to do is store the results in a blank text file: `conc_each_every_bnc_baby.txt`, which you have created beforehand and stored in `CLSR/chap5`.

```
> path.to.file <- "C:/CLSR/chap5/conc_each_every.txt" # Windows
> path.to.file <- "/CLSR/chap5/conc_each_every.txt" # Mac OS
> cat("file name\tsentence\tleft context\tnode\tright context", all.matches.tab.4, sep="\n",
file=path.to.file)
```

Once more, you should wait before congratulating yourself. When you open the text file with a spreadsheet software, the output is disappointingly messy. This is because there are sometimes several matches per line. For example, line 58 of `A39.xml` has two matches for *every*. The second match is appended at the end of the row. Ideally, the second match should have its own line. This is conceptually easy to fix: we need to repeat each sentence as many times as it has matches and delimit each match in turn. Technically, this is complex because it requires that we identify the position of each match in the sentence and tell R the beginning and the end of the match to insert tabulations accordingly.

5.2.2.2 Locating Substrings

We need to locate the position of identical matching patterns in a string. Gries (2009, pp. 138–140) offers a solution that involves a base R function: `gregexpr()`. This solution has been turned into a function: `exact.matches()`. Let me show you how it works.

Store sentence 58 from `A39.xml` in a character vector.

```
> A39s58 <- "Not every print, or every print-maker, is suitable for the purpose, however"
```

The first argument of `gregexpr()` is the pattern you want to match. The second argument is the string where the match is found. The remaining two arguments make the search case insensitive and compatible with regular expressions.

```
> gregexpr("every", A39s58, ignore.case=TRUE, perl=TRUE)
[[1]]
[1]  5 21
attr(,"match.length")
[1] 5 5
attr(,"useBytes")
[1] TRUE
```

The function outputs a list. The top-level index `[[1]]` indicates that the vector contains at least one match. The first lower-level index `[1]` indicates that we have two matches, whose starting character-wise positions are 5 and 21 respectively. The second lower-level positions indicate the length of each match: five characters. We can use this information to calculate the end position of each match.

Other functions from additional packages retrieve both the starting and ending positions of the matches. Such is the case of `stri_locate_all()` from the `stringi` package, `str_locate_all()` from the `stringr` package, and `substring.location()` from the `Hmisc` package. First, install and load the packages.

```
> # stringi
> install.packages("stringi")
> library(stringi)
>
> # stringr
> install.packages("stringr")
> library(stringr)
>
> # Hmisc
> install.packages("Hmisc")
> library(Hmisc)
```

Each function has its own argument order and specifications.

```
> # stringi
> stri_locate_all(pattern="every", A39s58, fixed = TRUE)
[[1]]
     start end
[1,]     5   9
[2,]    21  25
> # stringr
> str_locate_all(pattern="every", A39s58)
[[1]]
     start end
[1,]     5   9
[2,]    21  25
> # Hmisc
> substring.location(A39s58, "every")
$first
[1]  5 21

$last
[1]  9 25
```

All these functions output a list. To understand why this is relevant, let us load the whole corpus file `A3C.xml`.

```
> corpus.file <- scan("C:/CLSR/BNC_baby/A3C.xml", what="character", sep="\n") # Windows
> corpus.file <- scan("/CLSR/BNC_baby/A3C.xml", what="character", sep="\n") # Mac OS
```

Then we vectorize the search expression…

```
> expr.match <- "<w c5=\"(DT0|AT0)\" hw=\"(each|every)\" pos=\"(ADJ|ART)\">(each|every) ?</w>"
```

we isolate the sentences in the corpus file…

```
> sentences <- grep("<s n=", corpus.file, value=TRUE)
```

and we select those sentences that contain at least one match.

```
> sent.with.matches <- grep(expr.match, sentences, ignore.case=TRUE, perl=TRUE, value=TRUE)
```

For the remainder of the demonstration, we shall use the `stringi` package, which offers more options than `stringr` and is a nice alternative to `Hmisc` or base R `gregexpr()`. Thanks to `stri_locate_all()`, we can now retrieve the starting and ending positions of each match. We specify `regex=TRUE` for the function to understand the pattern of the match. We also specify

`opts_regex=stri_opts_regex(case_insensitive=TRUE)`

to make the search case insensitive.

```
> match.positions <- stri_locate_all(pattern=expr.match, sent.with.matches, regex=TRUE,
+                                    opts_regex=stri_opts_regex(case_insensitive=TRUE))
> length(match.positions) # find how many sentences contain at least one match
[1] 13
> head(match.positions, 3) # display the first 3 list elements (pay attention to the 2nd element)
[[1]]
     start end
[1,]   478 520

[[2]]
     start end
[1,]   325 365
[2,]   649 689
```

```
[[3]]
     start end
[1,]   637 677
```

A list is output (instead of a matrix or a data frame) because `lines.with.matches` is a thirteen-element vector. The function `stri_locate_all()` returns a result for each element in the vector. Since matches can be found several times in a same vector element (as is the case for the second sentence, which contains two matches), the output is more conveniently stored in a list.

Let us proceed with this file. In Sect. 5.2.2.1, our problem was that we obtained several tab-delimited matches in the same row when there were several matches per corpus line. To avoid this, we must repeat the corpus line as many times as it contains matches. The number of matches per line can be obtained with another `stringi` function: `stri_count()`.

```
> matches.per.sent <- stri_count(sent.with.matches, regex=expr.match,
+                                opts_regex=stri_opts_regex(case_insensitive=TRUE))
> matches.per.sent
 [1] 1 2 1 1 1 1 1 1 1 1 1 1 1
```

The second line contains two matches. Use `rep()` to repeat the lines as many times as needed.

```
> sent.per.match <- rep(sent.with.matches, matches.per.sent)
> length(sent.per.match)
[1] 14
```

The vector `sentences.per.match` now contains fourteen lines, because the corpus file contains fourteen matches.

The next step consists in iterating over the list elements to extract the starting and ending positions of each match thanks to a `for` loop. Once inside the loop, convert the list into a data frame so that each list element becomes a mini data frame. This makes the extraction much easier. Before you enter the loop, create two empty numeric vectors. They will be used to collect all the starting and ending positions at the end of the loop.

```
> # create two empty numeric vectors
> all.starts <- numeric()
> all.ends <- numeric()
>
> # enter the loop
> for (j in 1:length(match.positions)){
+
+   # convert the list into a data frame
+   match.positions.df <- as.data.frame(match.positions[[j]])
+
+   # extract the current starting positions
+   starts <- match.positions.df$start
+
+   # extract the current ending positions
+   ends <- match.positions.df$end
+
+   # collect all the starting and ending positions
+   all.starts <- c(all.starts, starts)
+   all.ends <- c(all.ends, ends)
+ } # exit the loop
```

When you inspect `all.starts` and `all.ends`, you can see that all fourteen starting and ending positions have been collected. The positions can be used as coordinated to retrieve exact matches from `sentences.per.match` with `substr()`. This function is used to extract or replace susbtrings in character strings. Its first argument is the string, its second argument is the starting position of the substring, and its third argument is the ending position of the substring.

```
> exact.matches <- substr(sent.per.match, all.starts, all.ends)
> exact.matches
 [1] "<w c5=\"AT0\" hw=\"every\" pos=\"ART\">every </w>"
 [2] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
 [3] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
 [4] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
 [5] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
 [6] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
 [7] "<w c5=\"AT0\" hw=\"every\" pos=\"ART\">every </w>"
 [8] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
 [9] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
[10] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
[11] "<w c5=\"DT0\" hw=\"each\" pos=\"ADJ\">each </w>"
[12] "<w c5=\"AT0\" hw=\"every\" pos=\"ART\">every </w>"
[13] "<w c5=\"AT0\" hw=\"every\" pos=\"ART\">every </w>"
[14] "<w c5=\"AT0\" hw=\"every\" pos=\"ART\">every </w>"
```

Although the output of `substr()` is neat, it is decontextualized and therefore not what we want. Tab-delimited matches are obtained by pasting five elements:

- the character string from the first position in `sentences.per.match` to the position right before the starting match position;
- a tabulation `"\t"`;
- the character string of the match (between the starting position and the ending position);
- a tabulation `"\t"`;
- the character string between the position right after the ending match position and the last position in `sentences.per.match`.

```
> delimited.matches <- paste(
+   substr(sent.per.match, 1, all.starts-1), "\t",
+   substr(sent.per.match, all.starts, all.ends), "\t",
+   substr(sent.per.match, all.ends+1, nchar(sent.per.match)),
+   sep="")
```

Time for cleanup.

```
> delimited.matches.clean <- gsub("<.*?>", "", delimited.matches, ignore.case=TRUE, perl=TRUE)
> delimited.matches.clean <- gsub(" *(\t) *", "\\1", delimited.matches.clean, perl=TRUE)
```

Inspect `delimited.matches.clean`. Even though the same sentence is repeated across the second and third elements of the vector, a distinct occurrence of *each* is tab-delimited each time.

5.2.2.3 The Full Script

To repeat the above steps with all the files of the BNC Baby, you need another `for` loop, referred to below as the first `for` loop. In this loop, make sure you tell R to go to the next corpus file if there is no match in a given corpus file.

```
> # clear R's memory
> rm(list=ls(all=TRUE))
>
> # load the stringi package
> library(stringi)
>
> # load the paths to all the corpus files
> corpus.files <- list.files(path="/CLSR/BNC_baby", pattern="\\.xml$", full.names=TRUE)
>
```

```
> # vectorize the search expression
> expr.match <- "<w c5=\"(DT0|AT0)\" hw=\"(each|every)\" pos=\"(ADJ|ART)\">(each|every) ?</w>"
>
> # prepare an empty vector to collect all matches at the end of the first for loop
> all.matches <- character()
>
> for (i in 1:length(corpus.files)){
+   # select the current corpus file (from 1 to 182)
+   corpus.file <- scan(corpus.files[i], what="char", sep="\n", quiet=TRUE)
+   # select the sentences in the current corpus file
+   sentences <- grep("<s n=", corpus.file, value=TRUE)
+   # select the sentences that contain at least one match
+   sent.with.matches <- grep(expr.match, sentences, ignore.case=TRUE,
+                             value=TRUE)
+   # get the number of matches per sentences
+   matches.per.sent <- stri_count(sent.with.matches, regex=expr.match,
+                                  opts_regex=stri_opts_regex(case_insensitive=TRUE))
+   # repeat the sentences as many times as there are matches
+   sent.per.match <- rep(sent.with.matches, matches.per.sent)
+   # locate the match positions
+   match.positions <- stri_locate_all(pattern=expr.match, sent.with.matches, regex=TRUE,
+                                      opts_regex=stri_opts_regex(case_insensitive=TRUE))
+
+   # if there are no matches, go the next corpus file
+   if (length(match.positions)==0) { next }
+
+   # if there are, prepare empty numeric vectors to store all match positions
+   all.starts <- numeric()
+   all.ends <- numeric()
+
+   # enter the second loop
+   for (j in 1:length(match.positions)){
+     match.positions.df <- as.data.frame(match.positions[[j]])
+     starts <- match.positions.df$start
+     ends <- match.positions.df$end
+     all.starts <- c(all.starts, starts)
+     all.ends <- c(all.ends, ends)
+   } # exit the second for loop
+
+   # tab-delimit the matches
+   delimited.matches <- paste(
+     substr(sent.per.match, 1, all.starts-1), "\t", # the left context
+     substr(sent.per.match, all.starts, all.ends), "\t", # the node
+     substr(sent.per.match, all.ends+1, nchar(sent.per.match)), # the right context
+     sep="") # an empty separator
+
+   # clean the delimited matches
+   delimited.matches.clean <- gsub("<.*?>", "", delimited.matches, ignore.case=TRUE)
+   delimited.matches.clean <- gsub(" *(\t) *", "\\1", delimited.matches.clean)
+
+   # prefix the name of the corpus file
+   delimited.matches.clean <- paste(basename(corpus.files[i]), delimited.matches.clean, sep="\t")
+
+   # collect all the cleaned-up tab-delimited matches
+   all.matches <- c(all.matches, delimited.matches.clean)
+ } # exit the second for loop
```

Once R has finished running the loop, store the results in a text file.

```
> cat("corpus file\tleft context\tnode\tright context", all.matches,
+     sep="\n", file="/CLSR/chap5/conc_bnc_baby.txt")
```

You may now congratulate yourself. The above script can be fine-tuned using a contextual window, as seen in Sect. 5.2.2.1.

5.3 Making a Data Frame from an Annotated Corpus

Once you have asked a theory-informed question and you have formulated and operationalized hypotheses, it is time to gather data. It is a good idea to include exact matches, i.e. concrete realizations in the corpus of the linguistic phenomenon that you study. You might recall Tab. 2.3 from Chap. 2, whose column `exact match` contains concrete uses of *quite* and *rather* in the BNC corpus. Let us learn how to make a similar data frame from the BNC Baby.

5.3.1 Planning the Data Frame

Let us keep things simple and retrieve only the following variables:

- the name of the corpus file;
- some information about the corpus file, namely the mode (written vs. spoken) and the genre (news, fiction, academic prose, etc.);
- the exact match (*each* or *every* in context);
- the determiner;
- the NP;
- the specific POS tag of the NP.

5.3.2 Compiling the Data Frame

To compile the data frame, we shall use the `strapplyc()` function from Gabor Grothendieck's `gsubfn` package, which you might recall from Sect. 4.4.9. Let us see how it works using a single file from the BNC Baby.

After clearing R's memory, we load the `gsubfn` package.

```
> rm(list=ls(all=TRUE))
> # install.packages("gsubfn")
> library(gsubfn)

Loading required package:  proto
```

We need to find a search expression for the matches. This expression has two parts, one that matches all instances of *each* and *every*,

```
<w c5="(DT0|AT0)" hw="(each|every)" pos="(ADJ|ART)">(each|every) ?</w>
```

and another that matches all noun phrases

```
<w c5="N[NP][012](-...)?" hw="\\w+" pos="SUBST">\\w+ ?</w>.
```

The BNC Baby tagset[2] tells you that four kinds of NPs are found in the corpus:

- common nouns, neutral for number, such as *aircraft* or *committee* (NN0),

[2] http://www.natcorp.ox.ac.uk/docs/c5spec.html.

- singular common nouns, such as *pencil* or *goose* (NN1),
- plural common nouns, such as *pencils* or *geese* (NN2), and
- proper nouns, such as *Kim* or *Elizabeth* (NP0).

The expression `N[NP][012]`, meaning "N followed by N or P, followed by 0 or 1 or 2", captures all kinds of NP tags. Also, bear in mind that the corpus contains ambiguity tags, such as "NN1-VVB" in `A3M.xml`:

```
<w c5="NN1-VVB" hw="stop" pos="SUBST">stop </w>.
```

The double tag indicates that the automatic tagger was unable to decide confidently which was the correct category of *stop*, NN1 (a singular common noun) or VVB (the finite base form of a lexical verb), and left two possibilities for the users to disambiguate. Appending `(-...)` to `N[NP][012]` guarantees that ambiguity tags are taken into account for all NPs.

Once you are happy with the search expression, vectorize it.

```
> expr.match <- "<w c5=\"(DT0|AT0)\" hw=\"(each|every)\" pos=\"(ADJ|ART)\">(each|every) ?</w>
+ <w c5=\"N[NP][012](-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>"
```

Load the corpus file.

```
> corpus.file <- scan("/CLSR/BNC_baby/A1E.xml", what="char", sep="\n")
```

Retrieve the classcode element from the corpus file as it contains all the information we need.

```
> classcode <- unlist(strapplyc(corpus.file, "<classCode scheme=\"DLEE\">(.*?)</classCode>"))
```

Retrieve the corpus file information.

```
> info <- unlist(strapplyc(corpus.file, "<[ws]text type=\"\\w+\">", backref=1))
```

From the `info` vector, extract the mode...

```
> mode <- unlist(strapplyc(info, "[ws]text"))
```

and the type.

```
> type <- unlist(strapplyc(info, "[ws]text type=\"(\\w+)\""))
```

It is now time to proceed with the search per se. Isolate the corpus sentences.

```
> sentences <- grep("<s n=", corpus.file, perl=TRUE, value=TRUE)
```

Get the matches with `strapplyc()`. Set `backref=0` to retrieve the full match.

```
> matches <- unlist(strapplyc(sentences, expr.match, ignore.case=TRUE, backref=0))
```

From the `matches` vector, extract the occurrences of *each* and *every*...

```
> determiner <- unlist(strapplyc(matches, "hw=\"(each|every)\""))
```

and the occurrences of NPs.

```
> NP_token <- unlist(strapplyc(matches, "pos=\"SUBST\">(\\w+) ?</w>"))
```

When you do not specify the `backref` argument of `strapplyc()`, you let the function determine the backreference automatically. Because only one element is bracketed above, the function returns the only available backreference. Regarding the POS tags of NPs, you cannot let `strapplyc()` decide automatically what to return because of the optional double tag between brackets. Enclose the part of the expression that you need between brackets and set `backref` to 1.

```
> NP_tag <- unlist(strapplyc(matches, "<w c5=\"(N[NP][012](-...)?)\"", backref=1))
```

It is now time to clean up the `matches` vector.

```
> # remove xml tags
> matches.clean <- gsub("<.*?>", "", matches, perl=T)
> # remove trailing spaces
> matches.clean <- gsub("< +$>", "", matches.clean, perl=T)
```

Finally, paste all the relevant elements in one row. The elements are separated by a tab so that each gets its own column.

```
> matches.row <- paste("A1E.xml", mode, type, matches.clean, determiner, NP_token, NP_tag, sep="\t")
> head(matches.row) # inspect
[1] "A1E.xml\twtext\tNEWS\teach nation\teach\tnation\tNN1"
[2] "A1E.xml\twtext\tNEWS\teach other\teach\tother\tNN1"
[3] "A1E.xml\twtext\tNEWS\teach other\teach\tother\tNN1"
[4] "A1E.xml\twtext\tNEWS\teach country \teach\tcountry\tNN1"
[5] "A1E.xml\twtext\tNEWS\teach type \teach\ttype\tNN1"
[6] "A1E.xml\twtext\tNEWS\tevery problem \tevery\tproblem\tNN1"
```

Upon inspection, the output is what we are looking for. We may run the script on all the corpus files in the BNC Baby thanks to a loop.

5.3.3 The Full Script

To repeat the above steps with all the files of the BNC Baby, you need a `for` loop. In this loop, make sure you tell R to go to the next corpus file if there is no match in a given corpus file. You also need to create an empty character vector to collect all matches at the end of the loop.

```
> # clear R's memory
> rm(list=ls(all=TRUE))
>
> # load the gsubfn package
> library(gsubfn)
>
> # load the paths to all the corpus files
> corpus.files <- list.files(path="/CLSR/BNC_baby", pattern="\\.xml$", full.names=TRUE)
>
> # vectorize the search expression
> expr.match <- "<w c5=\"(DT0|AT0)\" hw=\"(each|every)\" pos=\"(ADJ|ART)\">(each|every) ?</w>
+ <w c5=\"N[NP][012](-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>"
>
> # prepare an empty vector to collect all matches at the end of the first for loop
> all.matches <- character()
>
> # enter the loop
> for (i in 1:length(corpus.files)) {
+  # load current corpus file
+   corpus.file <- scan(corpus.files[i], what="char", sep="\n")
+   classcode <- unlist(strapplyc(corpus.file, "<classCode scheme=\"DLEE\">(.*?)</classCode>"))
```

```
+   # retrieve corpus file info
+  info <- unlist(strapplyc(corpus.file, "<[ws]text type=\"\\w+\">", backref=1))
+  # look for mode
+  mode <- unlist(strapplyc(info, "[ws]text"))
+  # look for type
+  type <- unlist(strapplyc(info, "[ws]text type=\"(\\w+)\""))
+
+  # isolate corp sentences and discard header
+   sentences <- grep("<s n=", corpus.file, perl=TRUE, value=TRUE)
+   # get matches (full pattern)
+  matches <- unlist(strapplyc(sentences, expr.match, ignore.case=TRUE, backref=0))
+  # if there are no matches, go to next corpus path ...
+  if (length(matches)==0) { next }
+
+  # collect the relevant elements
+  determiner <- unlist(strapplyc(matches, "hw=\"(each|every)\""))
+  NP_token <- unlist(strapplyc(matches, "pos=\"SUBST\">(\\w+) ?</w>"))
+  NP_tag <- unlist(strapplyc(matches, "<w c5=\"(N[NP][012](-...)?)\"", backref=1))
+
+  # clean up
+  matches.clean <- gsub("<.*?>", "", matches, perl=T) # remove xml tags
+  matches.clean <- gsub("< +$>", "", matches.clean, perl=T) # remove trailing spaces
+
+  # paste the collected elements
+  matches.row <- paste(basename(corpus.files[i]), classcode, mode, type, matches.clean,
+                       determiner, NP_token, NP_tag, sep="\t")
+
+  # collects all results
+  all.matches <- c(all.matches, matches.row)
+ } # exit the loop
```

Once R has finished running the loop, store the results in a text file.

```
> cat("corpus file\tinfo\tmode\ttype\texact match\tdeterminer\tNP\tNP_tag", all.matches, sep="\n",
+     file="C:/CLSR/chap5/df_each_every_bnc_baby.txt") # Windows
>
> cat("corpus file\tinfo\tmode\ttype\texact match\tdeterminer\tNP\tNP_tag", all.matches, sep="\n",
+     file="/CLSR/chap5/df_each_every_bnc_baby.txt") # Mac
```

Fig. 5.3 is what the file looks like when you open it with a spreadsheet software.

corpus file	info	mode	type	exact match	determiner	NP	NP_tag
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each nation	each	nation	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each other	each	other	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each other	each	other	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each country	each	country	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each type	each	type	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	every problem	every	problem	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each double	each	double	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	each jurisdiction	each	jurisdiction	NN1
A1E.xml	W newsp brdsht nat: commerce	wtext	NEWS	every share	every	share	NN1
A1F.xml	W newsp brdsht nat: editorial	wtext	NEWS	every day	every	day	NN1

Fig. 5.3: A snapshot of a data frame compiled from the BNC Baby

5.4 Frequency Lists

You generate a frequency list when you want to know how often words occur in a corpus. A frequency list is a two-column table with all word types occurring in the corpus in one column and their type frequency in the other column. A frequency list is useful when you want to know what a text is mostly about.

The principle of frequency lists is rooted is the distinction between type and token. The sentence *The cat is on the mat* contains six word tokens but five word types. This is because the sentence has six word occurrences, but only five different words. The word *the* occurs twice.

5.4.1 A Frequency List of a Raw Text File

Let us make a frequency list of Bram Stoker's *Dracula*. The good news is that we can recycle some of the code that we used in Sect. 5.2.1.

```
> # clear R's memory
> rm(list=ls(all=TRUE))
>
> # load the text
> dracula <- scan(file="/CLSR/chap5/dracula.txt", what="char", sep="\n")
> # you might need to add fileEncoding="UTF-8"
>
> # find the novel
> start <- grep("^CHAPTER I.$", dracula) ; start
[1] 132
> end <- grep("^\\/Jonathan Harker.\\/$", dracula) ; end
[1] 13014
> novel <- dracula[start:end]
>
> # collapse into a one-line text
> novel.vector <- paste(novel, collapse=" ")
```

What is new is the need to switch the whole novel to lowercase. If we do not, R is going to treat some words that we perceive as belonging to the same type (i.e. *The* and *the*) as two distinct types.

```
> novel.vector <- tolower(novel.vector)
```

Next, we split the vector at non-word characters with `strplit()`.

```
> split <- unlist(strsplit(novel.vector, "\\W+"))
```

Thanks to the `table()` function, we generate a table of frequencies and see how many times each word type occurs in the novel.

```
> table.split <- table(split)
> head(table.split)
split
 000    1   10   11   12 12th
   1   16    9    8    8    1
```

The point of making a frequency list is to see the most frequent items. Because the `table()` function sorts the values in alphabetic order, we should use `sort()` to sort the table in decreasing order.

```
> sorted.table.split <- sort(table.split, decreasing=TRUE)
> head(sorted.table.split, 15)
split
```

```
 the  and    i   to   of    a   he   in that   it
7876 5910 4843 4645 3612 2949 2583 2506 2486 2174
 was   as   we  for   is
1880 1592 1551 1533 1506
> tail(sorted.table.split, 15)
split
  yawing    yearn  yelling yellowed   yelpin
       1        1        1        1        1
 yelping     yews yielding     yoke     york
       1        1        1        1        1
       z     zeal  zealous      zoo zoophagy
       1        1        1        1        1
```

The fifteen most frequent words are not that interesting because they consist of closed-class words, i.e. words that serve a grammatical function. You have to go far down the list to find the first interesting open-class words: *time* and *(van) helsing*. This is bound to be the case whenever you make a frequency list from a natural language sample. Getting rid of grammatical words would have the advantage of focusing on open-class words (lexical words). The bottom of the list consists of unusual characters, rare words, or figures. These items tend to occur only once in the whole corpus. Word types illustrated by only one token are called *hapax legomena*. Often, corpus linguists get rid of them. However, depending on your research topic, they can bring useful information. I advise you to think twice before removing them.

5.4.1.1 Addressing Issues

We need to address three issues. First, although `strsplit()` does its job well, it does not tokenize the string well. Because we split at non-word characters, a string like "*can't*" was split into "*can*" and "*t*". Another option would have been to split at spaces. Although this would have worked well for "*can't*" (the negation marker would not stand on its own as *t*), that would leave us with strings like "*journal.*", "*whirlpool;*", "*paprika,*", or "*letter:–*", which include punctuation. This is not a limitation of `strsplit()` but rather an effect of the many rules and exceptions that you have to deal with when working with natural languages. You may of course address this issue in base R, but that requires recreating the rules and exceptions in a heuristic manner. Needless to say that this is a tedious procedure. A faster alternative is to use well-tested libraries to tokenize your character strings such as Stanford CoreNLP (Arnold and Tilton 2015, pp. 132–134).[3] Other tokenizers exist—e.g. `scan_tokenizer()` and `MC_tokenizer()` from the `tm` package—but they are subject to the same limitations as `strsplit()`.

The second issue is the presence of empty strings once you have split the text into words. Although unseen in this text file, it may arise depending on how you tokenize your character strings. For example, if you use the `MC_tokenizer()` function from the `tm` package on `novel.vector`, you will notice two empty character strings between *i* and *jonathan*. This is because the regular expression used to split the string ignored the punctuation in the text, but left blanks instead.

```
> library(tm)
> tokens <- MC_tokenizer(novel.vector)
> head(tokens)
[1] "chapter"  "i"        ""         ""
[5] "jonathan" "harker"
```

[3] See also http://stanfordnlp.github.io/CoreNLP/.

If you encounter this situation, you can easily get rid of empty strings by telling R to select those strings that are not blank thanks to `which()` and a logical operator.

```
> # get the positions of non empty strings
> tokens.not.empty <- which(tokens!="")
> #
> tokens <- tokens[tokens.not.empty]
> head(tokens)
[1] "chapter"  "i"        "jonathan" "harker"
[5] "s"        "journal"
```

Once this is done, you can proceed with `table()` and `sort()` to make the frequency list.

The third issue has to do with unwanted closed-class words. To get rid of them, it is customary to use a stoplist. A stoplist is a text file that contains those words that you do not want. You will find a stoplist for English in `CLSR/chap5`. The stoplist should match the language of your corpus and the period considered. Feel free to add or remove words from the stoplist as you see fit. Let us apply the stoplist to `sorted.table.split`. First, load the stoplist

```
> stoplist <- scan("/CLSR/chap5/english_stoplist.txt", what="character", sep="\n") # Windows
> stoplist <- scan("/CLSR/chap5/english_stoplist.txt", what="character", sep="\n") # Mac
```

Technically, `sorted.table.split` is a numeric array. The words can be obtained via the function `names()`. Knowing this, you can get rid of closed-class words contained in the stoplist by telling R to subset `sorted.table.split` and retain only those words that are not in the stoplist

```
> sorted.table.split.2 <- sorted.table.split[!names(sorted.table.split) %in% stoplist]
> head(sorted.table.split.2)
split
   time helsing     van   night    lucy    back
    386     322     322     316     299     262
```

The last step consists in converting the array into a tab-delimited frequency list that your spreadsheet software can display.

```
> freqlist <- paste(names(sorted.table.split.2), sorted.table.split.2, sep="\t")
```

You may now store the frequency list in a text file, e.g. `freqlist.dracula.txt`.

```
> # Windows
> cat("WORD\tFREQUENCY", freqlist, file="C:/CLSR/chap5/freqlist.dracula.txt", sep="\n")
> # Mac
> cat("WORD\tFREQUENCY", freqlist, file="/CLSR/chap5/freqlist.dracula.txt", sep="\n")
```

5.4.2 A Frequency List of an Annotated File

Frequency lists do not need to include all the words in a corpus. If you do not want grammatical words in the list, you can focus on nouns, general adjectives, and lexical verbs. Working with a POS-tagged corpus allows you to do it without a stoplist.

Before you start writing your search expressions to extract the nouns, the adjectives, and the verbs from the corpus, you should (again) consult the BNC Baby's tagset. Also, make sure you take the possibility of an ambiguity tag into account. Because we are interested in all kinds of nouns, we should write a regular expression that encompasses common nouns that are neutral for number (NN0), singular common nouns (NN1), plural common nouns (NN2), and proper nouns (NP0). Do not forget to escape the quotes.

```
<w c5=\"N[NP][012](-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>
```

The regular expression that matches all general adjectives (AJ0), as opposed to comparative (AJC) or superlative adjectives (AJS), is much simpler.

```
<w c5=\"AJ0(-...)?\" hw=\"\\w+\" pos=\"ADJ\">\\w+ ?</w>
```

Finally, because all the tags of lexical verbs start with VV, the regular expression should match all tags that start with two V and one word character.

```
<w c5=\"VV\\w(-...)?\" hw=\"\\w+\" pos=\"VERB\">\\w+ ?</w>
```

You are now ready to run the script, the first part of which is familiar to you (see Sect. 5.3.3).

```
> # clear R's memory
> rm(list=ls(all=TRUE))
>
> # load the gsubfn package
> library(gsubfn)
>
> # load the paths to all the corpus files
> corpus.files <- list.files(path="/CLSR/BNC_baby", pattern="\\.xml$", full.names=TRUE)
>
> # prepare an empty vector to collect all matches at the end of the first for loop
> all.matches <- character()
```

With a `for` loop, iterate over all the corpus files, look for the matches and collect the results. Nouns, adjectives and verbs being very frequent, you do not need the line of code that tells R what to do if no match is found.

```
> # enter the loop
> for (i in 1:length(corpus.files)) {
+   # load current corpus file
+   corpus.file <- scan(corpus.files[i], what="char", sep="\n")
+   # isolate corp sentences and discard header
+   sentences <- grep("<s n=", corpus.file, perl=TRUE, value=TRUE)
+   # collect the relevant elements
+   adjectives <- unlist(strapplyc(sentences,
+                                  "<w c5=\"AJ0(-...)?\" hw=\"\\w+\" pos=\"ADJ\">\\w+ ?</w>",
+                                  backref=0))
+   nouns <- unlist(strapplyc(sentences,
+                             "<w c5=\"N[NP][012](-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                             backref=0))
+   verbs <- unlist(strapplyc(sentences,
+                             "<w c5=\"VV\\w(-...)?\" hw=\"\\w+\" pos=\"VERB\">\\w+ ?</w>",
+                             backref=0))
+   # collect all matches
+   all.matches <- c(all.matches, adjectives, nouns, verbs)
+ } # exit the loop
```

The clean-up phase consists in removing the tags.

```
> # remove xml tags
> all.matches.clean <- gsub("<.*?>", "", all.matches, perl=T)
> head(all.matches.clean)
```

When you remove the tags, you preserve the spaces before or after the word. You should remove them.

```
> # remove leading and trailing spaces
> all.matches.clean <- gsub("(^ +| +$)", "", all.matches.clean, perl=TRUE)
> head(all.matches.clean)
```

You are now ready to tabulate,

```
> all.matches.table <- table(all.matches.clean)
> head(all.matches.table)
```

sort the results,

```
> all.matches.sorted.table <- sort(all.matches.table, decreasing=TRUE)
> head(all.matches.sorted.table)
```

and convert the array into a tab-delimited frequency list.

```
> tab.table <- paste(names(all.matches.sorted.table), all.matches.sorted.table, sep="\t")
> head(tab.table)
```

You may now save the frequency list in a text file, which your spreasheet software will display nicely.

```
> # Windows
> cat("WORD\tFREQUENCY", freqlist, file="C:/CLSR/chap5/freqlist.bnc.baby.txt", sep="\n")
> # Mac
> cat("WORD\tFREQUENCY", freqlist, file="/CLSR/chap5/freqlist.bnc.baby.txt", sep="\n")
```

Fig. 5.4 is what the file looks like when you open it in a spreasheet software. The top of the list consists of general verbs and nouns.

WORD	FREQUENCY
said	12704
know	10202
got	8825
get	6756
go	6427
think	5961
time	5827
see	5551
other	5448
want	4285
going	4144
way	3979
people	3846
good	3594
put	3575

Fig. 5.4: A snapshot of a frequency list of nouns, verbs, and adjectives in the BNC Baby

The frequency list you have just made consists of word tokens. The list is therefore crowded with a myriad of inflected forms. If you want a less crowded frequency list but do not want to compromise its lexical richness, you can modify the search expressions so that they return only lemmas. By way of illustration, *find* is the verbal lemma for verb forms such as *finds*, *finding*, and *found*; *car* is the nominal lemma for the noun

cars. To achieve this, just add a supplementary line when you collect nouns and verbs (English adjectives being invariable, no extra line of code is needed). Each time, select the head-word part of the tag that is prefixed by "hw=" (i.e. the lemma) and enclose the value between brackets. Setting `backref` to 0 will ensure that `strapplyc()` returns this value.

```
> # collect the relevant elements
> adjectives <- unlist(strapplyc(sentences,
+                                "<w c5=\"AJ0(-...)?\" hw=\"\\w+\" pos=\"ADJ\">\\w+ ?</w>",
+                                backref=0))
> nouns <- unlist(strapplyc(sentences,
+                           "<w c5=\"N[NP][012](-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                           backref=0))
> noun.lemmas <- unlist(strapplyc(nouns, "hw=\"(\\w+)\"", backref=1))
> verbs <- unlist(strapplyc(sentences,
+                           "<w c5=\"VV\\w(-...)?\" hw=\"\\w+\" pos=\"VERB\">\\w+ ?</w>",
+                           backref=0))
> verb.lemmas <- unlist(strapplyc(verbs, "hw=\"(\\w+)\"", backref=1))
> # collect all matches
> all.words <- c(all.words, adjectives, noun.lemmas, verb.lemmas)
```

Exercises

5.1. A concordance of an unannotated text
David Robinson has written the `gutenbergr` package which downloads and processes the public domain works in the Project Gutenberg database (Robinson 2016). The `gutenberg()` function allows you to kill two birds with one stone, namely (a) download one or more books by ID from the online database, and (b) strip out the header and footer. The package comes with an extensive data set of Project Gutenberg metadata. Use the package to make a concordance of *London* in Charles Dickens's *Great Expectations* (ID: 1400), *Bleak House* (ID: 1023), and *David Copperfield* (ID: 766). Choose a window of ten words before and after the match. It is a good idea to read the documentation first: https://cran.r-project.org/web/packages/gutenbergr/gutenbergr.pdf.

5.2. Making a data frame from an annotated corpus
Make a data frame, each line of which is an observation of the construction *it BE ADJ to V that* in the BNC Baby. The construction is exemplified in (2) and (3) below.

(2) **It is important to note that** these criteria are not intended to be an exhaustive definition of the public interest. (BNC–FRN)

(3) **It is plausible to assume that** case and thematic systems in these simple examples and also in more complicated cases are interrelated (...). (BNC–HJ1)

Each observation should be described by the following variables:

- `corpus file` the name of the file from the BNC Baby where the construction is observed;
- `info`: the information tag about the text (e.g. `W newsp brdsht nat: sports` or `W ac:polit law edu`);
- `mode`: `wtext` for written text or `stext` for spoken text;
- `type`: the text genre (`ACPROSE`, `NEWS`, `FICTION`, etc.);
- `exact match`: the construction in context;
- `ADJ`: the adjectival constituent of the construction;

- `V_inf`: the verbal constituent of the construction.

You are welcome to reuse the script from Sect. 5.3.3 and change only what is necessary.

5.3. Making a frequency list of bigrams in Melville's *Moby Dick*

An *n*-gram is a continuous sequence of *n* words from a character string. A unigram is an *n*-gram of size 1, a bigram is an *n*-gram of size 2, a trigram is an *n*-gram of size 3, etc. Let us take a simple example sentence:

There is a cat on the mat.

We find:

- Seven unigrams: *There*, *is*, *a*, *cat*, *on*, *the*, and *mat*.
- Six bigrams: *There is*, *is a*, *a cat*, *cat on*, *on the*, and *the mat*.
- Five trigrams: *There is a*, *is a cat*, *a cat on*, *cat on the*, and *on the mat*.

N-grams have a wide range of applications in NLP, the main one being probability models. Such models can be used to predict the probability of word sequences in a text. Such probabilities are commonly used in tasks like speech recognition, spelling correction, automatic completions, or machine translation, where words or word sequences have to be recognized from ambiguous inputs.

The purpose of this exercise is somewhat less ambitious. Your task is to:

- download Herman Melville's *Moby Dick* from Project Gutenberg using the `gutenbergr` package;
- make a frequency list of bigrams in the novel using the `ngram` package[4];
- use a stoplist to filter out unwanted bigrams.

References

Arnold, Taylor, and Lauren Tilton. 2015. *Humanities Data in R: Exploring Networks, Geospatial Data, Images, and Text*. Quantitative Methods in the Humanities and Social Sciences. New York: Springer.

Gries, Stefan Thomas. 2001. A Corpus-Linguistic Analysis of *-ic* and *-ical* Adjectives. *ICAME Journal* 25: 65–108. http://clu.uni.no/icame/ij25/gries.pdf.

Gries, Stefan Thomas. 2009. *Quantitative Corpus linguistics with R: A Practical Introduction*. New York, NY: Routledge.

Robinson, David. 2016. *gutenbergr: Download and Process Public Domain Works from Project Gutenberg*. R package version 0.1.2. https://CRAN.Rproject.org/package=gutenbergr.

[4] To access the documentation, enter `?ngram` after downloading and installing the package. To read the manual, visit https://cran.r-project.org/web/packages/ngram/ngram.pdf.

Chapter 6
Summary Graphics for Frequency Data

Abstract In this chapter, you will learn how to process frequency data and represent your findings graphically.

6.1 Introduction

Now that you know most of the character-string processing techniques, you are ready to extract data from corpora and obtain frequencies. The paragraphs below will show you how to summarize frequency data.

6.2 Plots, Barplots, and Histograms

In Chap. 4, you learned how to make a frequency list. You may want to see what the distribution of these words looks like. After loading the file `freqlist.bnc.baby.txt`, which you created previously, plot the frequencies with `plot()`. This generic function is one of the most widely used in R. It comes with many options, which I advise you to explore by entering `?plot`.

```
> rm(list=ls(all=TRUE)) # clear R's memory
> freqlist <- read.table("/CLSR/chap5/freqlist.bnc.baby.txt", header=TRUE) # load the freqlist
> str(freqlist) # inspect
> plot(freqlist$FREQUENCY, xlab="index of word types", ylab="frequency", cex=0.6)
```

Here, the `xlab` and `ylab` arguments specify the names of the horizontal and vertical axes respectively. The cex argument specifies the size of the circles that signal data points: `0.6` represents 60% of the default size (`1`). A title can also be added to the plot with `title`. Fig. 6.1a shows the resulting plot. Instead of circles, you may want to join the points with a line (Fig. 6.1b). If so, add the argument `type` and specify `"l"`. The argument `lwd=2` specifies the width of the line (`1` is the default).

```
> plot(freqlist$FREQUENCY, type="l", lwd=2, xlab="index of word types", ylab="frequency", cex=0.6)
```

Another option is to plot the words themselves. It would not make sense to plot all the words in the frequency list, so we subset the 20 most frequent words (Fig. 6.1c). First, create a new plot and specify

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_6

`col="white"` so that the data points are invisible. Next, plot the words with `text()`. Technically, the words are used as labels (hence the argument `labels`). These labels are found in the first column of the data frame: `freqlist$WORD`. To keep the labels from overlapping, reduce their size with `cex`.

```
> plot(freqlist$FREQUENCY[1:20], xlab="index of word types", ylab="frequency", col="white")
> text(freqlist$FREQUENCY[1:20], labels = freqlist$WORD[1:20], cex=0.7)
```

You can combine the labelling with the line by specifying `type="l"` in the `plot()` call.

```
> plot(freqlist$FREQUENCY[1:20], xlab="index of word types", ylab="frequency", type="l",
+      col="lightgrey")
> text(freqlist$FREQUENCY[1:20], labels = freqlist$WORD[1:20], cex=0.7)
```

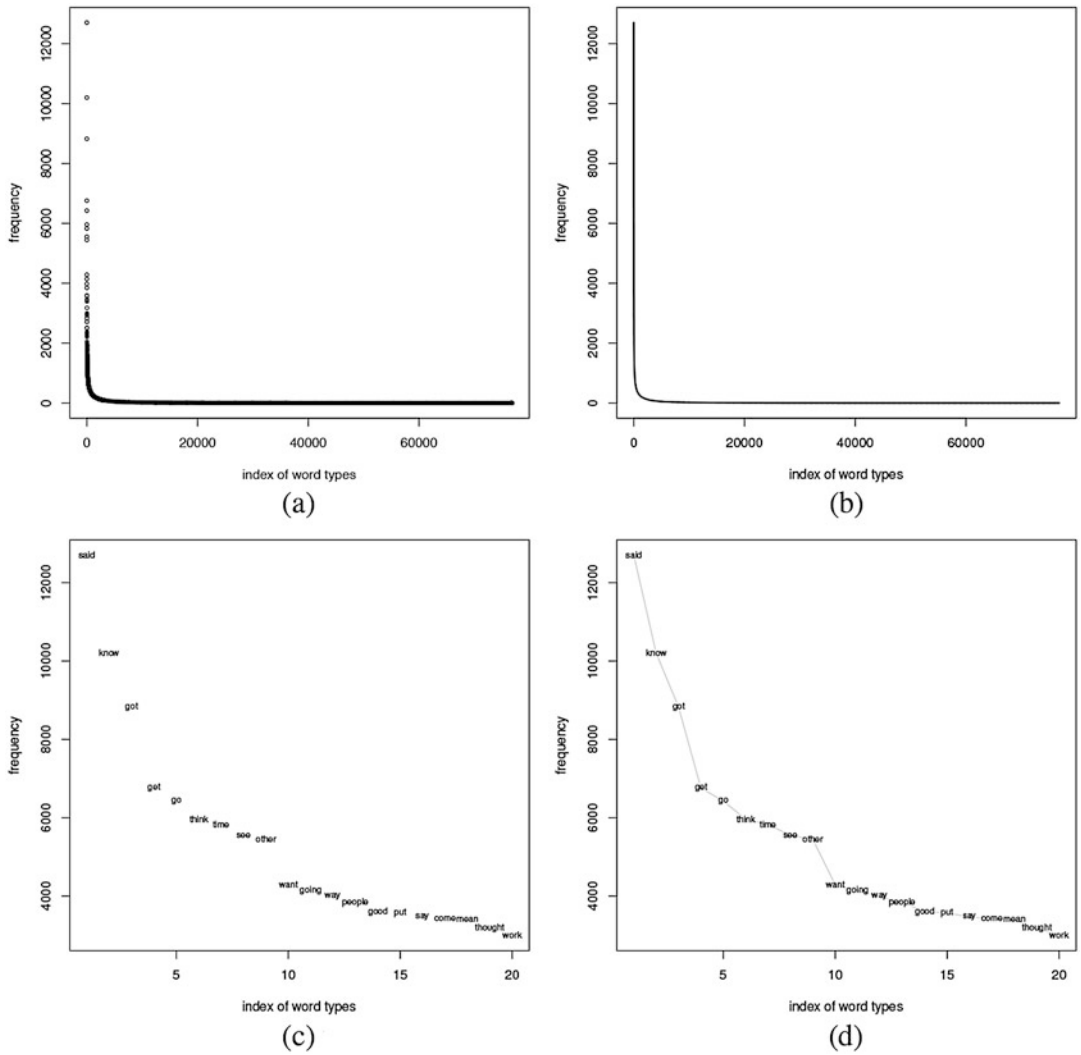


Fig. 6.1: Plotting a frequency list of the BNC Baby. (**a**) Points; (**b**) lines; (**c**) words; (**d**) words and line

The distribution at work is known as Zipfian. It is named after Zipf's law (see Sect. 8.8): many rare events coexist with very few large events. The resulting curve continually decreases from its peak (although, strictly speaking, this is not a peak). It is typical of the distribution of words in natural languages. If you plot the frequency list of any corpus of natural language, the curve will look invariably the same providing the corpus is large enough. Let us repeat the above with `freqlist.dracula.txt`.

```
> freqlist <- read.table("/CLSR/chap5/freqlist.dracula.txt", header=TRUE, row.names=1)
> plot(freqlist$FREQUENCY, type="l", lwd=2, xlab="index of word types", ylab="frequency", cex=0.6)
```

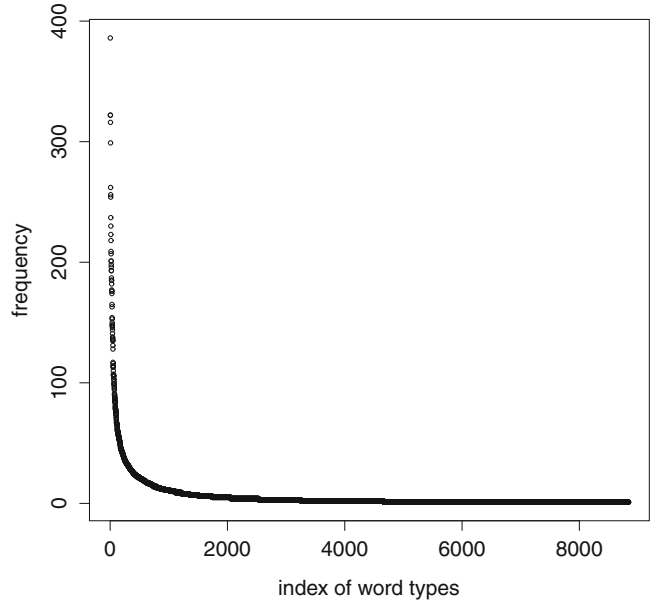


Fig. 6.2: Plotting a frequency list of Bram Stoker's *Dracula*

Again, the curve in Fig. 6.2 decreases steeply at first, and then slowly and continuously. A large number of rare (mostly lexical) words coexist with a few number of frequent (function) words.

Another way of plotting the data is by means of a barplot with the `barplot()` function. Fig. 6.3 plots the ten and twenty most frequent lexical words in the BNC Baby.

```
> barplot(freqlist$FREQUENCY[1:10],names.arg = freqlist$WORD[1:10], las=2)
> barplot(freqlist$FREQUENCY[1:20],names.arg = freqlist$WORD[1:20], las=2)
```

The heights of the bars in the plot are determined by the values contained in the vector `freqlist $FREQUENCY`. The argument `las` allows you to decide if the labels are parallel (`las=0`) or perpendicular (`las=2`) to the *x*-axis. Each bar represents a word type. The space between each bar indicates that these word types are distinct categories.

Histograms are close to barplots except that the bars do not represent distinct categories (therefore, there is no space between them). They represent specified divisions of the *x*-axis named "bins". Their heights are proportional to how many observations fall within them. As a consequence, increasing the number of observations does not necessarily increase the number of bins by the same number, as Fig. 6.4 shows.

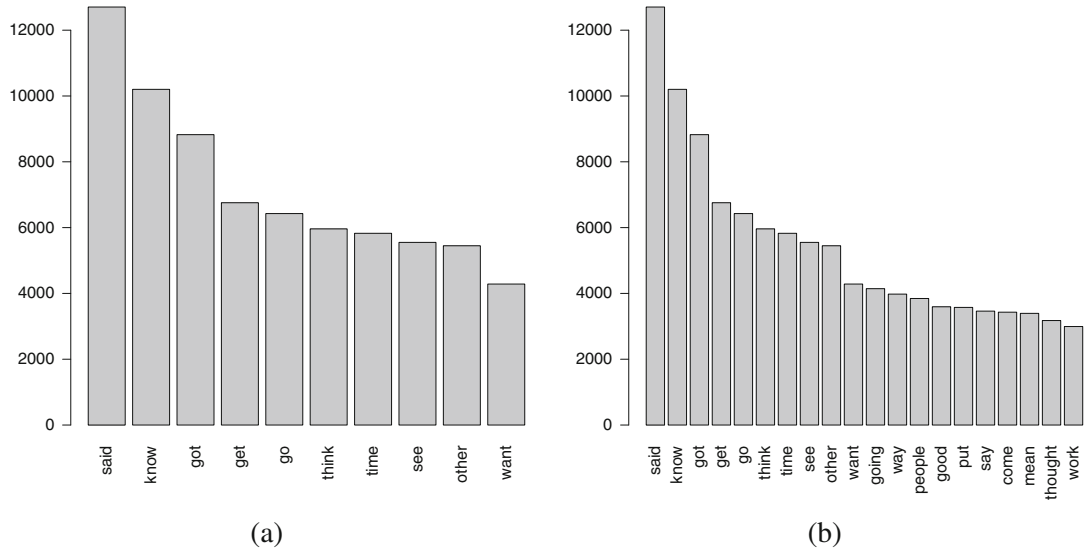


Fig. 6.3: Barplots of the most frequent words in the BNC Baby. (**a**) The ten most frequent lexical words in the BNC Baby. (**b**) the 20 most frequent lexical words in the BNC Baby

```
> hist(freqlist$FREQUENCY[1:10], xlab="frequency bins", las=2, main="") # 10 observations
> hist(freqlist$FREQUENCY[1:100], xlab="frequency bins", las=2, main="") # 100 observations
```

6.3 Word Clouds

Word clouds offer a trendy, user-friendly way of representing frequency lists graphically. A combination of two packages will do the trick: `tm` (see Sect. 5.4.1.1), and `wordcloud`. You should therefore install them before you proceed.[1] In the paragraphs below, you will learn how to make a word cloud based on Herman Melville's *Moby Dick* and another one based on the BNC Baby.

```
> install.packages("tm", repos = "http://cran.us.r-project.org")
> install.packages("wordcloud", repos = "http://cran.us.r-project.org")
```

The first step consists in placing the text which you want a word cloud of into a dedicated directory (e.g. `C:/CLSR/wordcloud/yourprojectname/yourtext.txt` for Windows users, or `/CLSR/wordcloud/yourprojectname/yourtext.txt` for Mac users). The directory `(C:)/CLSR/chap6/wordcloud/moby_dick` contains a text-file version of *Moby Dick*. The text file has been stripped of its line breaks.

[1] It is also a good idea to read the documentation of each package because the functions and their arguments are subject to change.

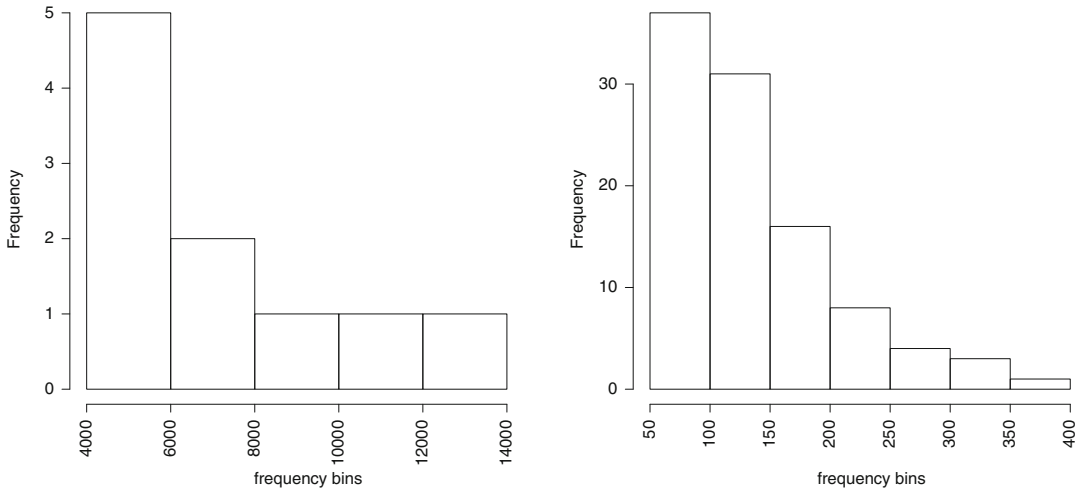


Fig. 6.4: Histograms of the most frequent words in the BNC Baby. (**a**) The ten most frequent lexical words in the BNC Baby. (**b**) The 100 most frequent lexical words in the BNC Baby

The second step consists in converting the text into a corpus so that the `tm` package can process it. In `tm` parlance, a corpus is a collection of documents, but it can also be a one-text corpus. This operation is done with the functions `Corpus()` and `DirSource()`. The argument of `DirSource()` is the path to the directory that contains the documents. Make sure that the directory contains only the files that you want the word cloud to be based on.

```
> rm(list=ls(all=TRUE))
> library(tm)

Loading required package:  NLP

> moby <- Corpus(DirSource("/CLSR/chap6/wordcloud/moby_dick"))
> # enter inspect(moby) to inspect
```

The `tm_map` function from the `tm` package contains a list of features meant to clean up the text. Most of these features should be familiar to you because they have been introduced in Chap. 4.

```
> moby <- tm_map(moby, stripWhitespace) # remove unnecessary white space
> moby <- tm_map(moby, content_transformer(tolower)) # convert everything to lower case
> moby <- tm_map(moby, removeNumbers) # remove digits
> moby <- tm_map(moby, removePunctuation) # remove punctuation
> moby <- tm_map(moby, removeWords, stopwords("english")) # use a built-in stoplist
```

You may now plot a word cloud of *Moby Dick* thanks to the `wordcloud()` function from the `wordcloud` package. The `max.words` argument is used to set the maximum number of words that you want to see plotted. The list of arguments to fine-tune the word cloud is large (see `?wordcloud`), so let me focus on the most important ones. The `min.freq` argument is used to select the minimal threshold below

which words are not plotted (here, the words whose frequency is below 50 do not appear in the cloud). In a word cloud, the size of a word is a function of its frequency: the more frequent a word, the larger its size. The words are plotted in decreasing frequency, the more frequent words being at the center of the plot, and the less frequent at the periphery. Likewise, the color scheme is based on frequency levels.

```
> library(wordcloud)
> moby <- Corpus(VectorSource(moby))
> wordcloud(moby, scale=c(5,0.5), max.words=100, min.freq=50, random.order=FALSE,
+           rot.per=.35, use.r.layout=FALSE, colors=brewer.pal(8, "Dark2"))
```

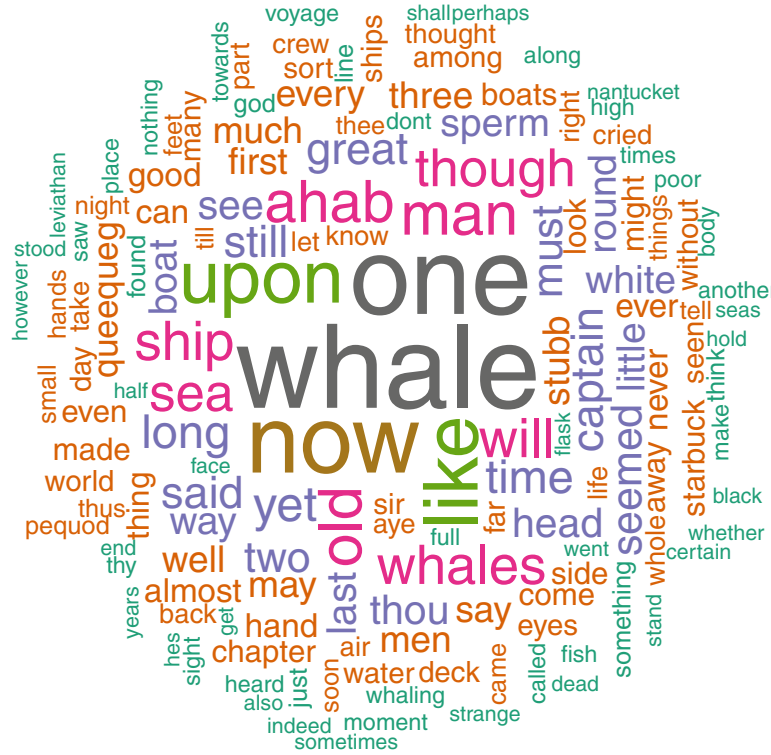


Fig. 6.5: A word cloud of Herman Melville's *Moby Dick*

In case you do not know what the novel is about, the word cloud in Fig. 6.5 offers a straightforward answer. The `tm_map` function has done a good job at making room for interesting lexical words to emerge (*whale(s)*, *Ahab*, *ship*, *sea*, etc. Note that such a good result is not always obtained, despite preprocessing the text prior to plotting the word cloud. This, I believe, is an effect of Herman Melville's careful planning and writing of the novel.

Frequency lists can also serve as input for word clouds. You do not need to run `tm_map` over the text to clean it up because the frequency list is based on a text that has already been preprocessed. However, the format of the frequency list needs to be altered slightly so that `tm` accepts it. You need to convert the two-column data frame where each word is assigned a numeric value denoting its frequency into a character

vector (space-separated, with no line break) in which each word is repeated as many times as it appears in the corpus. With respect to `freqlist`, this means that *said* will be repeated 12704 times, *know* 10202 times, *got* 8825 times, etc.

```
> rm(list=ls(all=TRUE)) # clear R's memory
> freqlist <- read.table("/CLSR/chap5/freqlist.bnc.baby.txt", header=TRUE) # load the freqlist
> head(freqlist)
   WORD FREQUENCY
1  said     12704
2  know     10202
3   got      8825
4   get      6756
5    go      6427
6 think      5961
```

The code below shows you how to do it. Only the 1000 most frequent words are kept.

```
> # repeat the word as many times as it appears in the corpus
> all.words <- as.vector(rep(freqlist$WORD, freqlist$FREQUENCY))
> # store the results in a text file
> cat(all.words, file="/CLSR/wordcloud/bncbaby/wcbnc.txt", sep=" ")
```

The resulting text file is stored in a dedicated directory whose path we use as input for the `DirSource()` function inside the `Corpus()` function.

```
> library(tm)
> bncbaby <- Corpus(DirSource("/CLSR/chap6/wordcloud/bncbaby"))
> inspect(bncbaby)
```

It is now time to run `wordcloud()` and plot the word cloud (Fig. 6.6).

```
> library(wordcloud)
> bncbaby <- Corpus(VectorSource(bncbaby))
> wordcloud(bncbaby, scale=c(5,0.5), max.words=150, min.freq=50,
+           random.order=FALSE, rot.per=.35, use.r.layout=FALSE,
+           colors=brewer.pal(8, "Dark2"))
```

Word clouds are useful when it comes to visualizing the output of topic modeling. Topic modeling consists in applying a statistical model to uncover the "hidden, abstract" concepts at work in a text (Jockers 2014, Chap. 13). In linguistics, however, word clouds are only as good as the frequency lists that they are based on. One major issue with frequency lists, and therefore word clouds, is that they give no indication as to the relationships between the words (antonymy, synonymy, hyponymy, etc.). I once plotted a word cloud based on the declarations of faith of the two remaining contenders in the second round of the French presidential election in 2012. The word cloud of the right wing contender gave pride of place to the word "socialist" (a left wing term). Anyone not knowing that the word was placed in the mouth of a right-wing contender would have believed, wrongly, that he was a socialist. The word cloud offered no way of knowing that, in fact, the right-wing contender's program hinged on attacking the opposing side (the Socialist Party).

Having said that, word clouds have some relevance in the framework of Frame Semantics. According to George Lakoff, if a political commentator asks a congressperson "Are you for or against tax relief?", she is framing taxes as an affliction. A liberal will not equate taxes with physical or moral pain since the more money a government obtains from taxes, the more they can spend on public services. When you use the phrase "tax relief", whether you are for or against it does not challenge the initial equation "taxes = affliction". Consequently, word clouds may be used to see how a text or corpus is framed, regardless of the semantic relationship between the words.

Fig. 6.6: A word cloud of the 150 most frequent words in the BNC Baby

6.4 Dispersion Plots

Frequency lists or word clouds aggregate the raw frequencies of linguistic units. As such, they provide a global measure of central tendencies at work across a corpus as a whole. While frequency lists are useful in many respects, they do not take into account the fact that tokens may be distributed unevenly in a corpus. The distribution of a linguistic unit across a corpus is known as its *dispersion*.

If you want to know exactly where different linguistic units tend to occur in a corpus or corpus file (for example to determine if a unit is equally spread throughout the corpus or occurs predominantly in one or more parts of the corpus), a dispersion plot is an obvious option. The paragraphs below explain how to plot the dispersion of a linguistic unit across a corpus file. Although dispersion plots are featured in ready-made software such as Wordsmith, I believe that it is a good thing to know how to make your own plots, so that you can fine-tune them and suit them to your needs. Below, I follow a solution proposed by Jockers (2014, Section 4.1) to study lexical dispersion in literary works. I adapt the proposal to plot the dispersion of linguistic units in an annotated corpus.

Our objective is to compare the dispersion of the noun types in one annotated file from the BNC Baby, namely `A1E.xml`. As you may remember from Sect. 5.3.2, there are four kinds of nouns in the BNC: common nouns, neutral for number (NN0), singular common nouns (NN1), plural common nouns (NN2), and proper nouns (NP0). We want one dispersion plot per noun type. I suggest we start with NN0. First, load the corpus file and isolate the sentences.

```
> # clear R's memory
> rm(list=ls(all=TRUE))
> # load the corpus file
> corpus.file <- scan("C:/CLSR/BNC_baby/A1E.xml", what="character", sep="\n") # Windows
> corpus.file <- scan("C:/CLSR/BNC_baby/A1E.xml", what="character", sep="\n") # Mac
> # select only corpus sentences and ignore the rest
> corpus.file.sentences <- grep("<s n=", corpus.file, perl=TRUE, value=TRUE)
```

To get the matches (and only the matches, not the full sentences), we use the `strapply()` function, from the `gsubfn` package.[2]

```
> # load the library which strapply() is part of
> library(gsubfn)

Loading required package:  proto

> # get all proper nouns
> tagged.NN0 <- unlist(strapply(corpus.file.sentences,
+                               "<w c5=\"NN0(-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                               perl=T, backref=1))
> # inspect
> head(tagged.NN0)
[1] "<w c5=\"NN0\" hw=\"697m\" pos=\"SUBST\">697m </w>"
[2] "<w c5=\"NN0\" hw=\"headquarters\" pos=\"SUBST\">headquarters</w>"
[3] "<w c5=\"NN0\" hw=\"staff\" pos=\"SUBST\">staff </w>"
[4] "<w c5=\"NN0\" hw=\"staff\" pos=\"SUBST\">staff </w>"
[5] "<w c5=\"NN0\" hw=\"staff\" pos=\"SUBST\">staff </w>"
[6] "<w c5=\"NN0\" hw=\"perch\" pos=\"SUBST\">perch </w>"
```

Now that you have collected all proper nouns in the file, it is time for cleanup.

```
> NN0 <- gsub("<.*?>", "", tagged.NN0, perl=T) # remove xml tags
> NN0 <- gsub("(^ | +$)", "", NN0, perl=T) # remove leading and trailing spaces
> head(NN0) # inspect
[1] "697m"         "headquarters" "staff"        "staff"        "staff"        "perch"
```

To generate a dispersion plot, we convert the corpus sentences into a linear flow of untagged words. We split up the character vector `corpus.file.sentences` at every occurrence of a tag with lazy matching,

```
> untagged.words <- unlist(strsplit(corpus.file.sentences, "<.*?>", perl=TRUE))
> head(untagged.words)
[1] ""           ""           "Latest "    ""           "corporate " ""
```

and we clean up the sentences.

```
> # remove selected punctuation
> untagged.words <- gsub("[,;:.?!-]", "", untagged.words)
> # remove leading and trailing spaces
> untagged.words <- gsub("(^ +| +$)", "", untagged.words, perl=TRUE)
> # get rid of empty character strings\index{character string}
> untagged.words <- untagged.words[nchar(untagged.words)>0]
> # inspect
> head(untagged.words)
[1] "Latest"    "corporate" "unbundler" "reveals"   "laidback"  "approach"
```

The *x*-axis denotes the sequence of untagged words. We create those values with `seq()` and store them in the numeric vector `seq.corpus`.

```
> seq.corpus <- seq(1:length(untagged.words))
```

Next, we determine the position of every occurrence of a number-neutral common noun in the vector `untagged.words`. We use the function `which()` (see Chaps. 2 and 4) to locate those words in

[2] If the corpus is large, and if there is a potentially high number of matches, I advise you to use `strapplyc()` instead or regular `strapply()`.

untagged.words that meet the condition "is a NN0 noun", and store them in a new numeric vector called NN0.positions. If we were looking for one specific noun (e.g. *cheese*), things would be easy because we would only need to type noun.positions <- which(untagged.words=="cheese"). Because the vector NN0 contains many different nouns, that will not work. We need a for loop. For each noun in the vector NN0, the loop retrieves the position of the noun in the untagged corpus file and stores the position in a vector. Then, it stores all positions of all nouns in the vector NN0.positions.

```
> # create an empty character vector to collect all results
> NN0.positions <- integer()
> # for each noun in the vector NN0...
> for (i in 1:length(NN0)) {
+   # get the position of the current NN0 in the untagged corpus file
+  NN0.position <- which(untagged.words==NN0[i])
+  # store the result in NN0.positions and start again
+  NN0.positions <- c(NN0.positions, NN0.position)
+ }
> # inspect
> head(NN0.positions)
[1]   13   97 1845 2106 2417 3909
```

Next, we need to address the issue of the values on the *y*-axis. We are not really counting number-neutral nouns, but merely assessing their presence or absence at each word position in the corpus file. Let 1 denote the presence of a NN0 noun and 0 its absence. We create a new vector named NN0.corpus where we will collect the matches later on. Right now, we fill this vector with as many NA values as there are positions in seq.corpus.[3]

```
> NN0.count <- rep(NA, length(seq.corpus))
```

Where a number-neutral noun is found, we replace NA with 1. We do this by selecting the positions in NN0.count that correspond to a NN0 noun. We use square brackets to do the subsetting.

```
> NN0.count[NN0.positions] <- 1
```

We know the positions where all number-neutral nouns are found. Each of these positions is assigned a value of 1. All the remaining positions are assigned a value of NA. We can finally plot the distribution of number-neutral common nouns across A1E.xml.

```
> plot(NN0.count, xlab="corpus file", ylab="common nouns", type="h", ylim=c(0,1), yaxt='n')
```

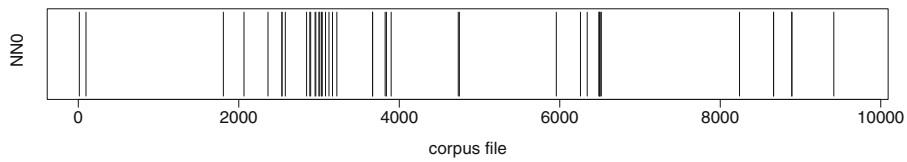


Fig. 6.7: Dispersion plot of number-neutral nouns in A1E.xml

This dispersion plot in Fig. 6.7 shows that the distribution of number-neutral nouns is extremely uneven across the whole corpus file. If you inspect the original corpus file, you see that NN0 is often used to code nouns that denote monetary values, as in (4):

[3] Depending on your experiment, NA stands for "not applicable", "not available", or "no answer".

(4) We have made a bid of nearly **£700m** for a company with a book value of **£200m**.

These values often appear in clusters, which explains their sporadic concentrations in the plot. After repeating the above steps for the three remaining noun types, we obtain Fig. 6.8.[4]

The distribution of common nouns in the singular (NN1) contrasts with that of number-neutral nouns. NN1 nouns are so frequent that their dispersion is homogeneous throughout the corpus file. Although common nouns in the plural and proper nouns are very frequent, their dispersions reveal concentrations (patches of black) and dilutions (patches of white).

6.5 Strip Charts

When you make a dispersion plot, each line represents one occurrence of the match. Strictly speaking, we are not so much counting occurrences as we are acknowledging their presence or absence. Most of the time, however, quantitative corpus linguistics consists in summing up frequencies and comparing frequency sums. When you compare frequency sums, you are interested in their distribution and, somehow, their dispersion. These can be visualized by means of a strip chart. Like dispersion plots, strip charts plot the data along an axis. Unlike dispersion plots, strip charts plot data sorted by value rather than by order of appearance.

The split infinitive is characterized by the insertion of an adverb between the infinitive marker *to* and the base form of the verb. The most famous example of a split infinitive is Captain Kirk's "To boldly go where no man has ever gone before", which was part of the original Star Trek introduction from 1966 onwards. The insertion of the adverb *boldly* between the marker *to* and the verb *go* caused quite a stir among prescriptivists at the time (ironically, the sexism of "no man" caused no such uproar). This usage is still branded as incorrect.

To illustrate the relevance of strip charts, load the `split_unsplit.rds` data frame.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("/CLSR/chap6/split_unsplit.rds") # Windows
> data <- readRDS("/CLSR/chap6/split_unsplit.rds") # Mac
```

The data frame compares the frequencies of two patterns in the Corpus of Historical American English (Davies 2010): <*to* ADV-*ly* V_{INF}> and <*to* V_{INF} ADV-*ly*> (ADV-*ly* stands for an adverb with a *-ly* ending). The former pattern illustrates the split infinitive whereas the second pattern illustrates the unsplit infinitive. The corpus is diachronic. Each line represents a decade, from `1810` to `2000`. The two columns provide the frequencies per decade for the split and unsplit variants. It is these two variables that we want to plot.

```
> str(data)
'data.frame': 20 obs. of  3 variables:
 $ DECADE            : int  1810 1820 1830 1840 1850 1860 1870 1880 1890 1900 ...
 $ SPLIT_INFINITIVE  : int  1812 1821 1836 1857 1875 1909 1991 2037 1999 2046 ...
 $ UNSPLIT_INFINITIVE: int  1851 2295 2726 2875 2931 2907 3157 3185 3180 3228 ...
```

Base R has a `stripchart()` function. If you plot one vector, specify this vector as the sole argument of the function (Fig. 6.9).

```
> stripchart(data$SPLIT_INFINITIVE)
```

[4] See Sect. A.1.1 for the full script.

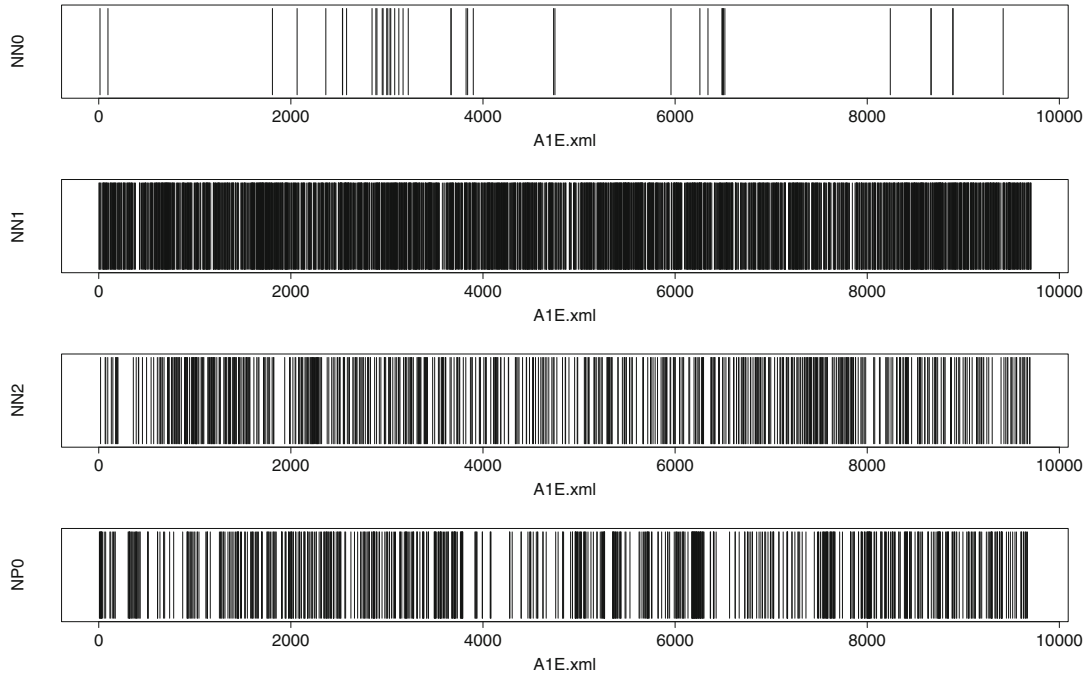


Fig. 6.8: Dispersion plots of noun types in `A1E.xml`

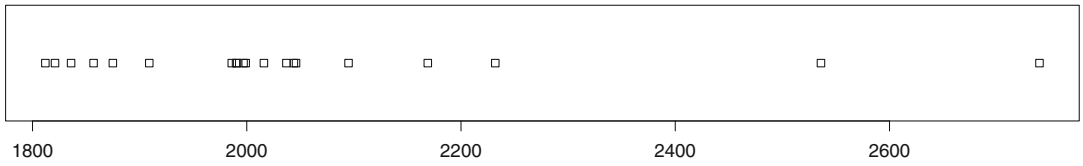


Fig. 6.9: A strip chart of the split infinitive in COHA

If you want to plot two or more vectors, you should group them in a list beforehand. When you do so, the name that you provide for each variable will be used to identify each strip chart in the final plot.

```
> x <- list("split"=data$SPLIT_INFINITIVE, "unsplit"=data$UNSPLIT_INFINITIVE)
```

You may now plot the two strip charts in the same plotting window (Fig. 6.10).

```
> stripchart(x, xlab="frequencies", ylab="construction")
```

You may want to customize the plot with graphic parameters such as the specification of one color per variable (`col = c("magenta", "darkblue")`). Because several data points overlap, you can add a small amount of noise to each variable to tease them apart with `jitter()` in the `method` argument of the function (Fig. 6.11).

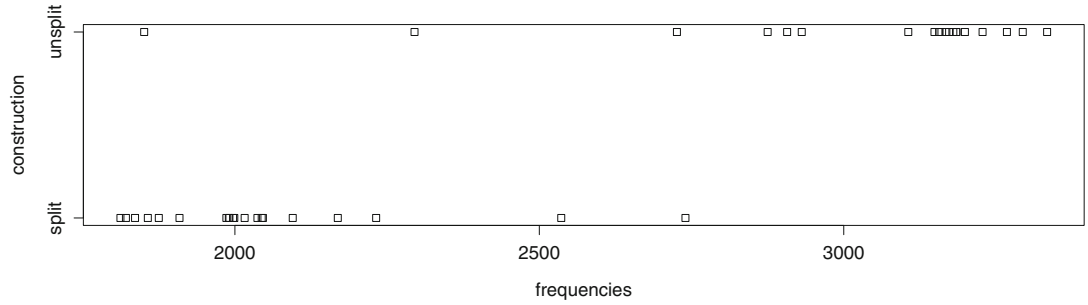


Fig. 6.10: A strip chart of <*to* ADV-*ly* V_{INF}> in COHA

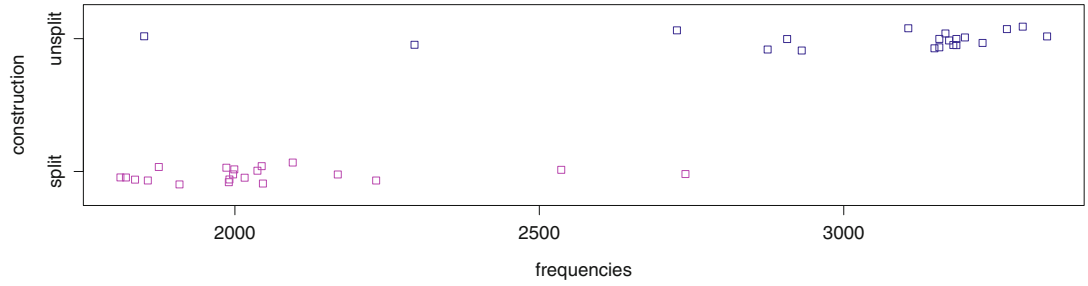


Fig. 6.11: The same strip chart with `jitter()`

```
> stripchart(x,
+   xlab = "frequencies",
+   ylab = "construction",
+   col = c("magenta", "darkblue"),
+   method = "jitter",
+ )
```

The respective frequency distributions of the split and unsplit infinitives differ. Not only is the unsplit infinitive more frequent than its split counterpart, but its frequencies also spread more along the range of values on the horizontal axis.

6.6 Reshaping Tabulated Data

So far, we have seen how to extract data from text-file corpora. Nowadays, however, online corpora abound. Not all of them allow you to easily export the data into a format that can be processed in R or a spreadsheet format. This is a shame because there is little point in designing powerful search engines and making them available on the Web if the user cannot further process the data. Fortunately, there are workarounds.

In Sect. 6.7 below, you will learn how to plot a motion chart of the split vs. unsplit infinitives in COHA across 20 decades with the `googleVis` package. This package requires that the frequency data be shaped in a specific format, as shown in Tab. 6.1. Because the corpus offers no easy way of exporting the frequency data, you have to reshape them so that they fit the package's requirements.

Table 6.1: Input frequency table for `GoogleVis`

adverb	decade	split infinitive	unsplit infinitive	total
actually	1810	0	0	0
actually	1820	0	0	0
actually	1830	0	18	18
actually	1840	0	63	63
actually	1850	0	18	18
actually	1860	0	18	18
actually	1870	0	27	27
actually	1880	1	9	10
actually	1890	0	45	45
actually	1900	1	36	37
actually	1910	2	27	29
actually	1920	1	72	73
actually	1930	2	0	2
actually	1940	0	18	18
actually	1950	2	45	47
actually	1960	3	36	39
actually	1970	4	0	4
actually	1980	8	0	8
actually	1990	29	0	29
actually	2000	37	27	64
...	...	...	...	...

First, retrieve the data from COHA, which consists of about 400 million word tokens and 115000 texts. The corpus is balanced by genre across twenty decades from the 1810s to the 2000s. It is ideal to see how American English has changed in the last two centuries. If you have not done it yet, visit `http://corpus.byu.edu/`, register (it is free), and login. Then, go to `http://corpus.byu.edu/coha/` and enter the corpus. This takes you to the query interface.

You have to run two queries. One for split infinitives and another for unsplit infinitives. To make the search easier, we shall restrict the extraction to adverbs in *-ly* (e.g. *boldly*, *really*, *absolutely*, etc.). A split infinitive will be defined as to followed by any *-ly* adverb, followed by the base form of any verb. An unsplit infinitive will be defined as to followed by the base form of any verb, followed by any *-ly* adverb. Now, we need to translate these definitions into something that the query form recognizes.

Queries made via the interface at `http://corpus.byu.edu/` follow a specific syntax.[5] Each part of speech has a code. To retrieve all kinds of adverbs, you should enter `[r*]`. To retrieve all adverbs that end in *-ly*, add `*ly` as a prefix (* stands for a wildcard, i.e. 'one or more word characters'). This gives you `*ly.[r*]`. To retrieve the base form of any verb, you should enter `[v?i*]`. When you combine these codes, you obtain the following:

- syntax for split infinitives: `to *ly.[r*] [v?i*]`
- syntax for unsplit infinitives: `to [v?i*] *ly.[r*]`

[5] `http://corpus.byu.edu/coha/help/syntax_e.asp`.

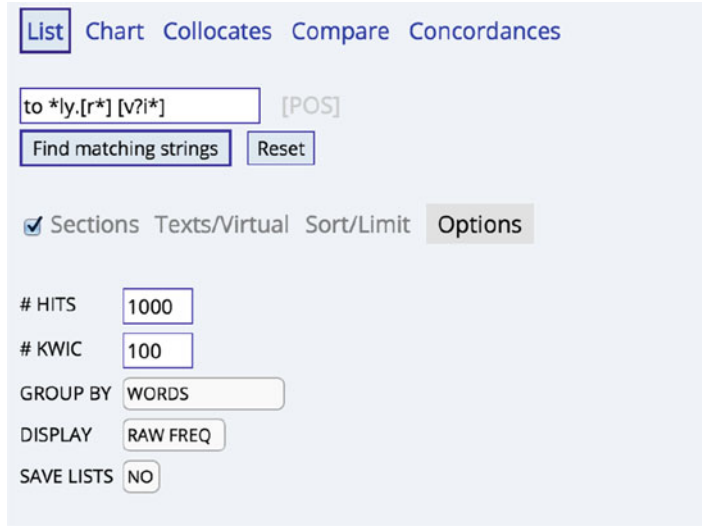
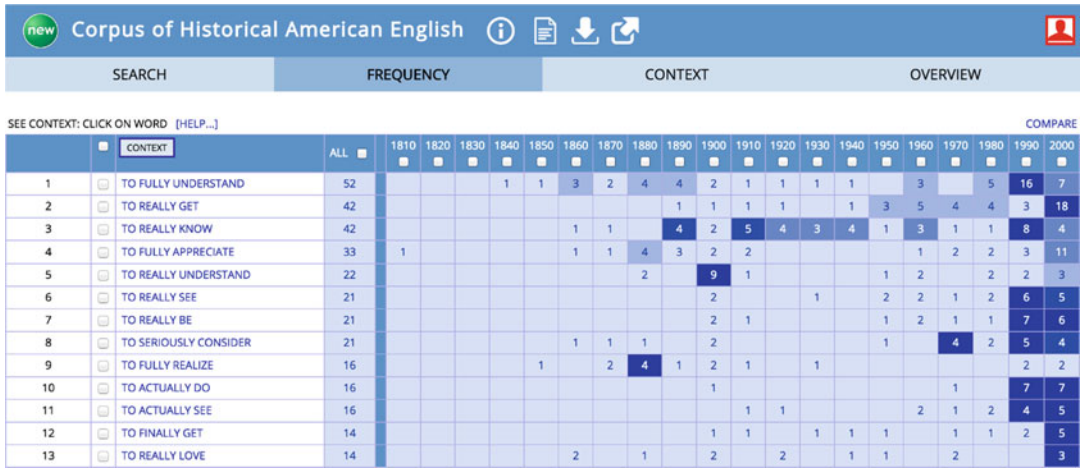


Fig. 6.12: Query on COHA

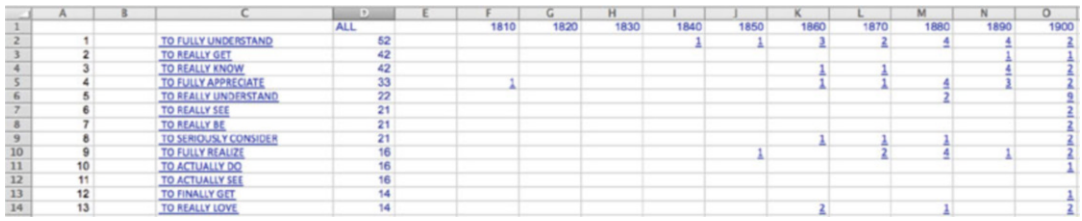


	CONTEXT	ALL	1810	1820	1830	1840	1850	1860	1870	1880	1890	1900	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000
1	TO FULLY UNDERSTAND	52				1	1	3	2	4	4	2	1	1	1	1		3		5	16	7
2	TO REALLY GET	42									1	1	1	1		1	3	5	4	4	3	18
3	TO REALLY KNOW	42						1	1		4	2	5	4	3	4	1	3	1	1	8	4
4	TO FULLY APPRECIATE	33	1					1	1	4	3	2	2					1	2	2	3	11
5	TO REALLY UNDERSTAND	22								2		9	1				1	2		2	2	3
6	TO REALLY SEE	21										2			1		2	2	1	2	6	5
7	TO REALLY BE	21										2	1				1	2	1	1	7	6
8	TO SERIOUSLY CONSIDER	21						1	1	1		2					1		4	2	5	4
9	TO FULLY REALIZE	16					1		2	4	1	2	1		1						2	2
10	TO ACTUALLY DO	16										1							1		7	7
11	TO ACTUALLY SEE	16											1	1				2	1	2	4	5
12	TO FINALLY GET	14										1	1		1	1	1		1	1	2	5
13	TO REALLY LOVE	14						2		1		2		2		1	1		2			3

Fig. 6.13: Query results

Let us start with the split infinitive. The pattern `to *ly.[r*] [v?i*]` should be entered in the blank search box (Fig. 6.12). You should also click "Options" and set "# HITS" to 1000 instead of 100. Tick the "Sections" box and click on "Find matching strings". Leave the other options unchanged. After a few seconds, the results are displayed (Fig. 6.13). You want to copy and paste the data from the online output into a spreadsheet. Here is a quick and dirty way: click somewhere in the upper left part of the window and drag down to the bottom right part. Press CONTROL + A (or CMD + A if you are on a Mac). Paste the selection into a spreasheet (CONTROL + V or CMD + V). As you can see from Fig. 6.14, the ouput of cut and paste is messy and there is some manual clean-up to do. Note that the output depends on what browser you use.

In your spreadsheet software, delete unwanted rows and columns (do not forget to delete the row total at the bottom of the table) and shift the column header to the right so that each decade is on top of its own column. Replace each empty cell with `0`. Assign the name `WORD` to the column that contains the matches. In this column, we only want the adverb, so we have to delete `TO` and the verb. This can be done easily in a text editor that implements regular expressions (Notepad++ for Windows or BBedit/TextWrangler for Mac). Copy and paste the whole table into your text editor. Activate the find-and-replace function by entering CONTROL/CMD + F and make sure the grep option is active. To remove `TO` and the verb, replace `TO (\w+LY) \w+` with `\1`.[6] When you deal with the unsplit infinitive, replace `TO \w+ (\w+LY)` with `\1`. To remove all remaining spaces (including trailing spaces), replace `|^\s` with an empty character, i.e. nothing. Note that there is a space before the pipe symbol |.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1				ALL		1810	1820	1830	1840	1850	1860	1870	1880	1890	1900
2	1		TO FULLY UNDERSTAND	52					1	1	3	2	4	4	2
3	2		TO REALLY GET	42										1	1
4	3		TO REALLY KNOW	42							1	1		4	2
5	4		TO FULLY APPRECIATE	33		1					1	1	4	3	2
6	5		TO REALLY UNDERSTAND	22									2		9
7	6		TO REALLY SEE	21											2
8	7		TO REALLY BE	21											2
9	8		TO SERIOUSLY CONSIDER	21							1	1	1		2
10	9		TO FULLY REALIZE	16						1		2	4	1	2
11	10		TO ACTUALLY DO	16											1
12	11		TO ACTUALLY SEE	16											
13	12		TO FINALLY GET	14											1
14	13		TO REALLY LOVE	14							2		1		2

Fig. 6.14: What the output looks like in Excel (snapshot)

Once you are satisfied with the clean-up, save the text file using an obvious filename, e.g. `split_inf.txt`. Repeat the above steps for the unsplit infinitive. Once you are done, you should have two distinct text files: `split_inf.txt` and `unsplit_inf.txt`.

There are two remaining issues:

- your data consists of two data frames instead of one;
- your two data frames do not have the format of Tab. 6.1.

In other words, you must go from two data frames in which each decade is a variable to a single data frame where decades are the factors of a single variable. Two R packages come to the rescue: `reshape` and `plyr`.

Install and load the `reshape` package.

```
> install.packages("reshape")
> library(reshape)
```

Load `split_infinitive.txt`, which is available from your `CLSR` folder.

```
> split <- read.delim("/CLSR/chap6/split_infinitive.txt", header=TRUE, check.names=FALSE)
> str(split) # inspect
'data.frame': 1000 obs. of  21 variables:
 $ WORD: Factor w/ 202 levels "ABRUPTLY","ABSOLUTELY",..: 67 125 125 67 125 125 125 139 67 5 ...
 $ 1810: int  0 0 0 1 0 0 0 0 0 0 ...
 $ 1820: int  0 0 0 0 0 0 0 0 0 0 ...
 $ 1830: int  0 0 0 0 0 0 0 0 0 0 ...
 $ 1840: int  1 0 0 0 0 0 0 0 0 0 ...
 $ 1850: int  1 0 0 0 0 0 0 0 1 0 ...
 $ 1860: int  3 0 1 1 0 0 0 1 0 0 ...
 $ 1870: int  2 0 1 1 0 0 0 1 2 0 ...
 $ 1880: int  4 0 0 4 2 0 0 1 4 0 ...
```

[6] In most programming languages other than R, one backslash is enough to escape characters.

```
 $ 1890: int  4 1 4 3 0 0 0 0 1 0 ...
 $ 1900: int  2 1 2 2 9 2 2 2 2 1 ...
 $ 1910: int  1 1 5 2 1 0 1 0 1 0 ...
 $ 1920: int  1 1 4 0 0 0 0 0 0 0 ...
 $ 1930: int  1 0 3 0 0 1 0 0 1 0 ...
 $ 1940: int  1 1 4 0 0 0 0 0 0 0 ...
 $ 1950: int  0 3 1 0 1 2 1 1 0 0 ...
 $ 1960: int  3 5 3 1 2 2 2 0 0 0 ...
 $ 1970: int  0 4 1 2 0 1 1 4 0 1 ...
 $ 1980: int  5 4 1 2 2 2 1 2 0 0 ...
 $ 1990: int  16 3 8 3 2 6 7 5 2 7 ...
 $ 2000: int  7 18 4 11 3 5 6 4 2 7 ...
```

With `reshape`, we "melt" the data so that each row is a unique word-decade-frequency combination. This is done with the `melt()` function.

```
> split.melt <- melt(split, id="WORD")
> str(split.melt) # inspect
'data.frame': 20000 obs. of  3 variables:
 $ WORD    : Factor w/ 202 levels "ABRUPTLY","ABSOLUTELY",..: 67 125 125 67 125 125 125 139 67 5 ...
 $ variable: Factor w/ 20 levels "1810","1820",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ value   : int  0 0 0 1 0 0 0 0 0 0 ...
```

We make the names of the second and third columns more explicit.

```
> colnames(split.melt)[2] <- "DECADE" # change name of 2nd column
> colnames(split.melt)[3] <- "SPLIT_INF"# change name of 3rd column
> head(split.melt, 10) # inspect
        WORD DECADE SPLIT_INF
1      FULLY   1810         0
2     REALLY   1810         0
3     REALLY   1810         0
4      FULLY   1810         1
5     REALLY   1810         0
6     REALLY   1810         0
7     REALLY   1810         0
8  SERIOUSLY   1810         0
9      FULLY   1810         0
10  ACTUALLY   1810         0
```

We repeat the above steps with `unsplit_infinitive.txt`

```
> unsplit <- read.delim("/CLSR/chap6/unsplit_infinitive.txt", header=TRUE, check.names=FALSE)
> unsplit.melt <- melt(unsplit, id="WORD") # melt
> colnames(unsplit.melt)[2] <- "DECADE" # change name of 2nd column
> colnames(unsplit.melt)[3] <- "UNSPLIT_INF" # change name of 3rd column
> head(unsplit.melt, 10) # inspect
         WORD DECADE UNSPLIT_INF
1        ONLY   1810           0
2    ENTIRELY   1810           6
3   PERFECTLY   1810           5
4       FULLY   1810           2
5      REALLY   1810           1
6      MERELY   1810           0
7  ABSOLUTELY   1810           2
8      EASILY   1810           0
9      WHOLLY   1810           1
10    EQUALLY   1810           3
```

Next, merge the two data frames into a unique data frame with `merge()`, a function from base R.

```
> df <- merge(split.melt, unsplit.melt, by=c("DECADE","WORD")) # merge
> str(df)
'data.frame': 111100 obs. of  4 variables:
 $ DECADE     : Factor w/ 20 levels "1810","1820",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ WORD       : Factor w/ 202 levels "ABRUPTLY","ABSOLUTELY",..: 2 2 2 3 3 3 3 3 3 3 ...
 $ SPLIT_INF  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ UNSPLIT_INF: int  2 0 0 0 0 0 0 0 0 0 ...
```

Upon inspection, we see that DECADE is not numeric. We convert the factor into a numeric vector.

```
> df$DECADE <- as.numeric(as.character(df$DECADE)) # set DECADE to numeric
```

The last step is to summarize the merged data frame with Hadley Wickham's plyr package.

```
> install.packages("plyr")
> library(plyr)
```

The ddply() function applies functions for each subset of the data frame and combines the results into a new data frame. Inside the ddply() call, the summarise function performs group-wise summaries.

```
> df <- ddply(df, c("WORD","DECADE"), summarise, SPLIT_INF=sum(SPLIT_INF),
+             UNSPLIT_INF=sum(UNSPLIT_INF)) # summarize
> head(df) # inspect
       WORD DECADE SPLIT_INF UNSPLIT_INF
1 ABSOLUTELY   1810         0           2
2 ABSOLUTELY   1820         0           3
3 ABSOLUTELY   1830         0          13
4 ABSOLUTELY   1840         0          11
5 ABSOLUTELY   1850         0           5
6 ABSOLUTELY   1860         0           6
```

Finally, we add a total column for combined frequencies.

```
> df$TOTAL_FREQ <- df$SPLIT_INF + df$UNSPLIT_INF
> head(df, 10) # inspect
         WORD DECADE SPLIT_INF UNSPLIT_INF TOTAL_FREQ
1  ABSOLUTELY   1810         0           2          2
2  ABSOLUTELY   1820         0           3          3
3  ABSOLUTELY   1830         0          13         13
4  ABSOLUTELY   1840         0          11         11
5  ABSOLUTELY   1850         0           5          5
6  ABSOLUTELY   1860         0           6          6
7  ABSOLUTELY   1870         0          22         22
8  ABSOLUTELY   1880         3          20         23
9  ABSOLUTELY   1890         3          26         29
10 ABSOLUTELY   1900         0          26         26
```

6.7 Motion Charts

In 2006, Hans Rosling gave a TED talk on social and economic developments in the world over the past 50 years.[7] The talk was a landmark in the TED talks series not only because of its contents, but also because of the clarity with which the data was visualized. Rosling used animated bubble charts. Inspired by Rosling's presentation (and other similar talks) Gesmann and de Castillo (2011) developed the googleVis package. The package allows R and the Google visualization API to communicate to create interactive charts.

In diachronic linguistics, the best known of these charts is the motion chart, popularized by Martin Hilpert. Hilpert (2011) compares motion charts to flipbooks that allow you to see how a given process of language change deploys over time.[8]

[7] https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen.

[8] http://members.unine.ch/martin.hilpert/motion.html.

Thanks to the `googleVis` package, we can plot a motion chart based on the data frame that we compiled in the previous section (`df`) with only two lines of code (four if you count the lines used to install and load the package).

```
> install.packages(googleVis)
> library(googleVis)
```

First, prepare a motion chart object with the `gvisMotionChart()` function. The variable that we are interested in is `WORD` (i.e. the adverbs that split or do not split the infinitive). The time variable is `DECADE`. We provide these variables via the function's arguments `idvar` and `timevar`.

```
> motionc.object <- gvisMotionChart(df, idvar="WORD", timevar="DECADE")
```

Finally, we plot the motion chart with `plot()`.

```
> plot(motionc.object)
```

The above line opens up a window in your browser displaying the motion chart (Fig. 6.15). Before you click on play, you may want to adjust playback speed (it is the button to the right of the main play button). You may also want to change the display from raw frequencies (Lin) to log-transformed values (Log): this has the effect of spreading the data points and making tendencies easier to spot (Fig. 6.15c and 6.15d).

Each bubble stands for an adverb. The size of the bubble depends on the adverb's combined frequency in both split and unsplit infinitive constructions. The position of the adverb indicates how frequently it occurs in the split (*x* axis) or unsplit (*y* axis) infinitive. The closer an adverb gets to the diagonal from the lower-left corner to the upper-right corner of the plot, the more compatible it is with both constructions. The further right an adverb goes, the stronger its preference for the split construction. The further up an adverb goes, the stronger its preference for the unsplit construction (Fig. 6.15b).

Unsurprisingly, the unsplit infinitive is far more frequent than its split counterpart. Nevertheless, a handful of adverbs (such as *really*, *fully*, *only*, and *carefully*) make occasional inroads into split constructions (e.g. 1860s–1890s). The years around 1900 are the most conservative. The situation changes slightly after World War II. Disappointingly, the airing of Star Trek does not seem to have any visible impact around 1966. *Really* and *fully* stand out as the boldest adverbs: they lead the trend from the 1980s onwards. *Boldly*, on the other hand, does not bulge. The toy dataset that I used has limitations (small size, only one adverb type, etc.), and the corpus is certainly biased with respect to genre. Therefore, we should be wary of making final conclusions based on this motion chart alone. Having said that, motion charts work great as a first approach. For more details regarding the code behind each motion chart, see this Google Developers page https://developers.google.com/chart/interactive/docs/gallery/motionchart.

Exercises

6.1. Barplots

The data file for this exercise is `modals.by.genre.BNC.rds`. You will find it in `(C:)/CLSR/chap6`. The data table shows the distribution of ten English modals across eight text genres in the BNC. Write a loop that makes a barplot for each modal in the table.

6.2. Dispersion plots

Make a dispersion plot for each of the following place names in Dickens's *Oliver Twist*, whose Project Gutenberg ID is 730: *Clerkenwell*, *Snow Hill*, *Holborn*, *Jacob's Island*, *Southwark*, *London Bridge*, *Newgate*, *Pentonville*, and *Smithfield*. Arrange the eight plots should in five rows and two columns.

6.3. Strip charts

The data file for this exercise is the same as for Exercise 6.1, namely: `modals.by.genre.BNC.rds`. Compare the distribution of modals across the eight text genres with a strip chart. To do so, you will need to transpose the data frame. The modals should be on the *y*-axis and their frequencies on the *x*-axis.

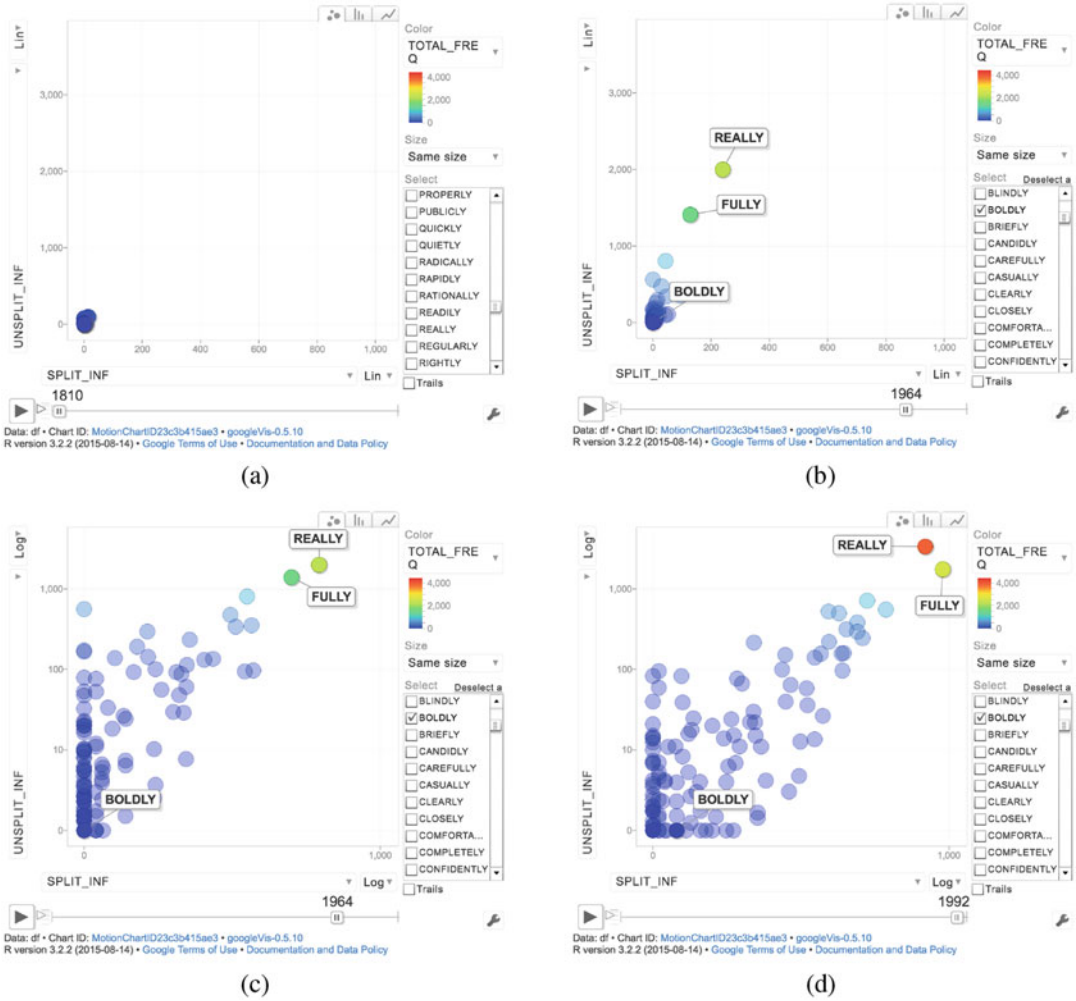


Fig. 6.15: A motion chart of the split vs. unsplit infinitive in COHA. (**a**) Initial state (raw frequencies); (**b**) 1964 (raw frequencies); (**c**) 1964 (logged); (**d**) 1992 (logged)

References

Davies, Mark. 2010. *The Corpus of Historical American English: 400 Million Words, 1810–2009*. http://corpus.byu.edu/coca/

Gesmann, Markus, and Diego de Castillo. 2011. Using the Google Visualisation API with R. *The R Journal* 3 (2): 40–44. https://journal.r-project.org/archive/2011-2/RJournal_2011-2_Gesmann+de~Castillo.pdf

Hilpert, Martin. 2011. Dynamic Visualizations of Language Change: Motion Charts on the Basis of Bivariate and Multivariate Data from Diachronic Corpora. *International Journal of Corpus Linguistics* 16 (4): 435–461. http://dx.doi.org/10.1075/ijcl.16.4.01hil. http://www.jbe-platform.com/content/journals/10.1075/ijcl.16.4.01hil

Jockers, Matthew. 2014. *Text Analysis with R for Students of Literature*. New York: Springer.

Part II
Statistics for Corpus Linguistics

Just like not all corpus linguists engage in some form of quantification, not all corpus linguists engage in statistics. Often, I hear colleague confess that they would like to use statistics in their research, but they refrain from doing it for lack of training in the field and for fear of an insuperable learning curve. Admittedly, linguists tend to receive very little training in statistics, and learning these techniques is challenging. However, let us face it: contemporary linguistics is at yet another quantitative turn in its history. Graduate programs throughout the world dramatically improve their offers in statistical training. Massive data sets are piling up. To achieve far-reaching results, the discipline needs robust statistical tools.

Some other colleagues also fear that, by doing statistics, they will be perceived as a mathematicians or "language engineers", not as linguists anymore. This is a blunt misconception. Doing statistics is useless unless driven by solid theory. In other words, you cannot make a good use of statistics in linguistics if you do not have a strong background in theoretical linguistics.

There are, obviously, wrong reasons for doing statistics when you are a linguist: tables, charts, plots, and *p*-values look nice in a paper; you want your research to be 100% objective; it is trendy. Most statistics are used for a main reason: you want to determine whether your observations are due to chance or not. If they are not, congratulations, you may have discovered something significant. While subjectivity is the key component of a linguist's job, a certain dose of objectivity is beneficial at certain steps of a linguist's work (see Fig. 1.1 from Sect. 1).

Chapter 7
Descriptive Statistics

Abstract Descriptive statistics summarize information. In this chapter, we review two kinds of descriptive statistics: measures of central tendency and measures of dispersion. Measures of central tendency are meant to summarize the profile of a variable. Although widespread, these statistics are often misused. I provide guidelines for using them. Measures of dispersion are complementary: they are meant to assess how good a given measure of central tendency is at summarizing the variable.

7.1 Variables

A variable is a property that varies from one individual to another. An individual may be just anything, such as a person (a speaker, an informant) or a linguistic unit or phenomenon (modal auxiliaries, transitivity, etc.). Here is an open-ended list of variables for linguistic phenomena:

1. the number of modal auxiliaries per sentence,
2. the number of syllables per word,
3. vowel lengths in milliseconds,
4. pitch frequencies in hertz,
5. ranked acceptability judgments,
6. the text types represented in a corpus,
7. the verb types that occur in a construction.

The first five of these variables provide numerical information and are known as quantitative variables. The last two variables provide non-numerical information and are known as qualitative or categorical variables.

Quantitative variables break down into discrete and continuous variables. Typically, discrete quantitative variables involve counts (integers). Such is the case of the number of modal auxiliaries per sentence and the number of syllables per word. Continuous quantitative variables involve a measurement of some kind within an interval of numbers with decimals. Such is the case of vowel lengths in milliseconds and pitch frequencies in hertz. There is a special type of quantitative variables: ordinal variables. Ordinal variables are numerical and take the form of rankings. An example of ordinal variable is the ranking of agentivity criteria: "1" high agentivity, "2" mild agentivity, "3" low agentivity. Acceptability judgments in psycholinguistic experiments

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_7

may also be ordinal variables. The specificity of ordinal variables is that you cannot do arithmetic with them because the difference between each level is not quantitatively relevant.[1]

Qualitative variables are nominal or categorical. They are usually coded with a label. For example, a verb type may be coded using POS tags, such as VVI for the infinite form of lexical verbs in the BNC or VVN for the past participle form. Although they are sometimes coded with numbers (e.g. "1" for infinitives, and "2" for past participles), you cannot do arithmetic with them. In R, you should always declare these numbers as characters (in vectors) or factors (in data frames).

7.2 Central Tendency

Measures of central tendency summarize the profile of a variable. There are three measures of central tendency: the mean, the median, and the mode. All three measures can summarize a large data set with just a couple of numbers.

7.2.1 The Mean

The mean (also known as the arithmetic mean) gives the average value of the data. The data must be on a ratio scale. The arithmetic mean (μ), is the sum ($\sum$) of all the scores for a given variable (x) in the data divided by the number of values (N):

$$\mu = \frac{\sum x}{N}. \tag{7.1}$$

Let us go back to the data set `split_unsplit.rds`.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("/CLSR/chap6/split_unsplit.rds") # Windows
> data <- readRDS("/CLSR/chap6/split_unsplit.rds") # Mac
```

Upon inspection, the variable `DECADE` is recognized as a numeric vector. Strictly speaking, it should be a factor because its values are nominal. Although not a crucial step with respect to the statistics that follow, conversion is advised.

```
> str(data)
'data.frame': 20 obs. of  3 variables:
 $ DECADE           : int  1810 1820 1830 1840 1850 1860 1870 1880 1890 1900 ...
 $ SPLIT_INFINITIVE  : int  1812 1821 1836 1857 1875 1909 1991 2037 1999 2046 ...
 $ UNSPLIT_INFINITIVE: int  1851 2295 2726 2875 2931 2907 3157 3185 3180 3228 ...
> data$DECADE <- as.factor(data$DECADE)
```

R has a built-in function to compute the mean of a numeric vector: `mean()`. To know the mean number of split and unsplit infinitives across the whole period, we apply the function to each vector.

```
> mean(data$SPLIT_INFINITIVE)
[1] 2049.4
> mean(data$UNSPLIT_INFINITIVE)
[1] 3018.35
```

[1] It would make little sense to say that mild agentivity is twice as less as high agentivity.

On average, there are 2049.4 occurrences of the split infinitive and 3018.35 occurrences of the split infinitive. These values represent only a summary of the data. No decade displays either figure. To visualize where the mean stands in your data, plot the numeric vectors and position the mean as a horizontal line with `abline()`. The `h` argument specifies where the horizontal line that corresponds to the mean should be plotted on the *y* axis. You obtain Fig. 7.1.[2]

```
> par(mfrow=c(1,2))
> plot(data$SPLIT_INFINITIVE)
> abline(h = mean(data$SPLIT_INFINITIVE), col="blue")
> text(5, mean(data$SPLIT_INFINITIVE)+15, "mean", col="blue")
> plot(data$UNSPLIT_INFINITIVE)
> abline(h = mean(data$UNSPLIT_INFINITIVE), col="green")
> text(5, mean(data$UNSPLIT_INFINITIVE)+20, "mean", col="green")
```

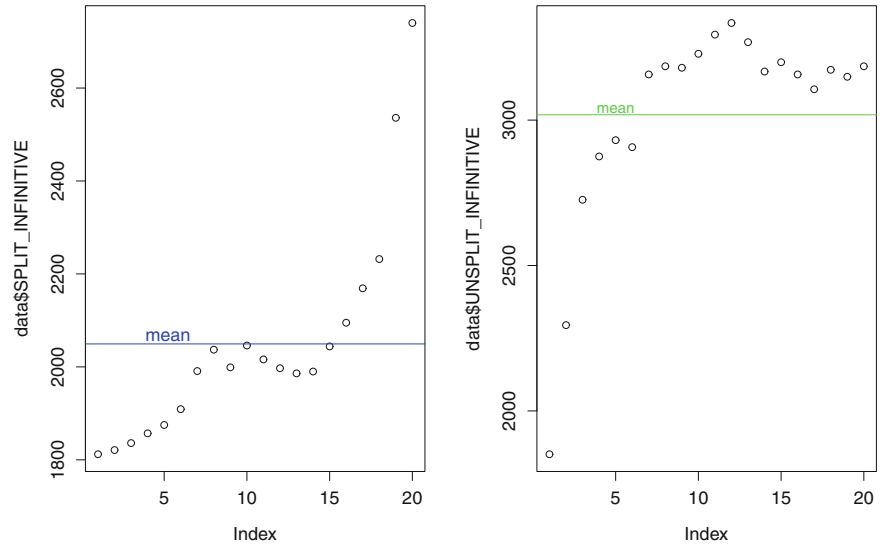


Fig. 7.1: Plotting the means of `SPLIT_INFINITIVE` and `UNSPLIT_INFINITIVE`

Interestingly, the median corresponds to the mean of the two middle values if the data consists of an even number of values.

```
> a <- 1:12; a
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> median(a)
[1] 6.5
> mean(c(6,7))
[1] 6.5
```

Although very popular among linguists, the mean is far from reliable. Consider the two vectors below.

[2] With `par()`, you can set up graphical parameters. A layout with two plots side by side is specified using `mfrow()`. The line `par(mfrow=c(1,2))` means "**m**ulti**f**rame, **row**-wise, 1 line × 2 columns layout". As a result, the two plots are organized in one row and two columns.

```
> b <- c(10, 30, 50, 70, 80)
> c <- c(10, 30, 50, 70, 110)
```

The vectors `b` and `c` are identical except for one value. In corpus linguistics, this might be caused by a word whose frequency is abnormally high. This minor difference translates into a large difference in the mean because of the few data that we have.[3]

```
> mean(b)
[1] 48
> mean(c)
[1] 54
```

To address this problem. The `mean()` function has an optional argument, `trim`, which allows you to specify a proportion of outlying values that are removed from the computation of the mean. Each vector contains five values. If you want to remove the top and bottom values, you need to set `trim` to 0.2 (i.e. 20%). Because $5 \times 0.2 = 1$, setting `trim` to 0.2 will remove two values in each vector: the highest value and the lowest value.

```
> mean(b, trim = 0.2) # = mean(c(30, 50, 70))
[1] 50
> mean(c, trim = 0.2) # = mean(c(30, 50, 70))
[1] 50
```

The resulting trimmed means are equal. Trimming means makes sense if the data set is large. If it is not, you should reconsider calculating the mean, whether trimmed or not.

7.2.2 The Median

The median is the value that you obtain when you divide the values in your data set into two "equal" parts. When your data set consists of an uneven number of values, the median is the value in the middle. In the vector `b`, this value in the middle is `50`. We can verify this with the `median()` function.

```
> median(b)
[1] 50
```

there is an equal number of values on either part of the median. When the vector consists of an even number of values, the value in the middle does not necessarily correspond to a value found in the vector.

```
> median(c(b, 100))
[1] 60
```

As opposed to the mean, the median is not affected by extreme values. What the median does not tell you is the behavior of the values on either side of it.

```
> par(mfrow=c(1,2))
> plot(data$SPLIT_INFINITIVE)
> abline(h = mean(data$SPLIT_INFINITIVE), col="blue")
> abline(h = median(data$SPLIT_INFINITIVE), col="blue", lty=3)
> text(5, mean(data$SPLIT_INFINITIVE)+15, "mean", col="blue")
> text(5, median(data$SPLIT_INFINITIVE)+15, "median", col="blue")
```

[3] As a rule, the smaller the data set, the more sensitive it is towards extreme values.

```
> plot(data$UNSPLIT_INFINITIVE)
> abline(h = mean(data$UNSPLIT_INFINITIVE), col="green")
> abline(h = median(data$UNSPLIT_INFINITIVE), col="green", lty=3)
> text(5, mean(data$UNSPLIT_INFINITIVE)+20, "mean", col="green")
> text(5, median(data$UNSPLIT_INFINITIVE)+20, "median", col="green")
```

As Fig. 7.2 shows, the median does not necessarily correspond to the mean.

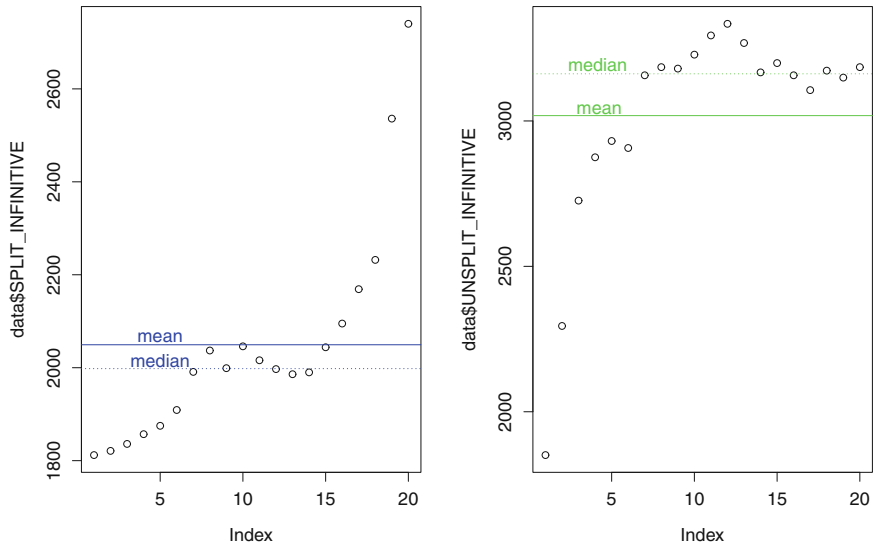


Fig. 7.2: Plotting the medians of SPLIT_INFINITIVE and UNSPLIT_INFINITIVE

7.2.3 The Mode

The mode is the nominal value that occurs the most frequently in a tabulated data set. You obtain the mode with `which.max()`. With respect to `data$SPLIT_INFINITIVE` or `data$SPLIT_INFINITIVE`, it does not make sense to determine the mode because each frequency value occurs once,

```
> tab <- table(data$SPLIT_INFINITIVE); tab

1812 1821 1836 1857 1875 1909 1986 1990 1991 1997 1999 2016 2037 2044 2046 2095 2169 2232 2536 2740
   1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
```

so let us come back to `df_each_every_bnc_baby.txt`, a data frame that we compiled in Chap. 4.

```
> df.each.every <- read.delim("C:/CLSR/chap5/df_each_every_bnc_baby.txt", header=TRUE) # Windows
> df.each.every <- read.delim("/CLSR/chap5/df_each_every_bnc_baby.txt", header=TRUE) # Mac
```

We want to know the mode of the variable `NP_tag`, so that we know which value is most often observed among the four possible options (NN0, NN1, NN2, and NP0). We isolate and tabulate the variable of interest with `table()`.

```
> str(df.each.every)
'data.frame': 2339 obs. of  8 variables:
 $ corpus.file: Factor w/ 159 levels "A1E.xml","A1F.xml",..: 1 1 1 1 1 1 1 1 1 2 ...
 $ info       : Factor w/ 22 levels "S conv","W ac:humanities arts",..: 10 10 10 10 10 10 10 10 10 11 ...
 $ mode       : Factor w/ 2 levels "stext","wtext": 2 2 2 2 2 2 2 2 2 2 ...
 $ type       : Factor w/ 5 levels "ACPROSE","CONVRSN",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ exact.match: Factor w/ 912 levels "each action ",..: 252 267 267 97 449 785 136 207 819 592 ...
 $ determiner : Factor w/ 2 levels "each","every": 1 1 1 1 1 2 1 1 2 2 ...
 $ NP         : Factor w/ 597 levels "act","action",..: 334 356 356 127 554 407 173 275 463 147 ...
 $ NP_tag     : Factor w/ 6 levels "NN0","NN1","NN1-AJ0",..: 2 2 2 2 2 2 2 2 2 2 ...
> tab.NP.tags <- table(df.each.every$NP_tag)
```

We run `which.max()` on the tabulated data. The function returns the mode and its position.

```
> which.max(tab.NP.tags)
NN1
  2
```

The mode of `NP_tag` is NN1. The mode is the tallest bar of a barplot (Fig. 7.3).

```
> barplot(tab.NP.tags)
```

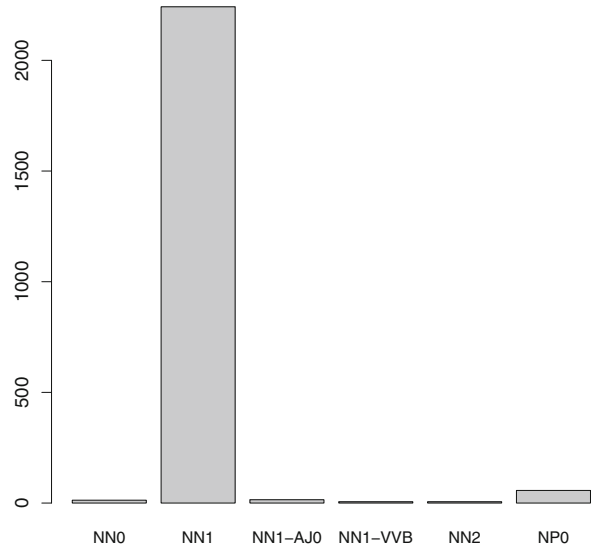


Fig. 7.3: A barplot showing the mode of `NP_tag`

The corresponding frequency of the mode is given by `max()`.

```
> max(tab.NP.tags)
[1] 2242
```

7.3 Dispersion

Dispersion is the spread of a set of observations. If many data points are scattered far from the value of a centrality measure, the dispersion is large.

7.3.1 Quantiles

By default, the `quantile()` function divides the frequency distribution into four equal ordered subgroups known as quartiles. The first quartile ranges from 0% to 25%, the second quartile from 25% to 50%, the third quartile from 50% to 75%, and the fourth quartile from 75% to 100%.

```
> quantile(data$SPLIT_INFINITIVE, type=1)
  0%  25%  50%  75% 100%
1812 1875 1997 2046 2740
> quantile(data$UNSPLIT_INFINITIVE, type=1)
  0%  25%  50%  75% 100%
1851 2907 3157 3185 3334
```

The `type` argument allows the user to choose from nine quantile algorithms, the detail of which may be accessed by entering `?quantile`. By default, R uses the seventh type.

The interquartile range (IQR) is the difference between the third and the first quartiles, i.e. the 75th and the 25th percentiles of the data. It may be used as an alternative to the standard deviation to assess the spread of the data (see Sect. 7.3.3 below).

```
> IQR(data$SPLIT_INFINITIVE, type=1) # = 2046 - 1875
[1] 171
> IQR(data$UNSPLIT_INFINITIVE, type=1) # = 3185 - 2907
[1] 278
```

The IQR values confirm what we already know: the frequency distribution of the split infinitive is less dispersed than the frequency distribution of the unsplit infinitive.

In base R, the `summary()` function combines centrality measures and quantiles (more precisely quartiles).

```
> summary(data$SPLIT_INFINITIVE)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1812    1900    1998    2049    2058    2740
> summary(data$UNSPLIT_INFINITIVE)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1851    2925    3162    3018    3188    3334
```

The same function can be applied to the whole data frame.

```
> summary(data)
     DECADE   SPLIT_INFINITIVE UNSPLIT_INFINITIVE
 1810   : 1   Min.   :1812     Min.   :1851
 1820   : 1   1st Qu.:1900     1st Qu.:2925
 1830   : 1   Median :1998     Median :3162
 1840   : 1   Mean   :2049     Mean   :3018
 1850   : 1   3rd Qu.:2058     3rd Qu.:3188
 1860   : 1   Max.   :2740     Max.   :3334
 (Other):14
```

7.3.2 Boxplots

In Sects. 6.4 and 6.5, dispersion plots and strip charts have given us the opportunity to appreciate dispersion graphically. In practice, a boxplot provide a more elaborate graphic representation of the spread of the values around a central point (Fig. 7.4). A boxplot is the graphic equivalent of `summary()`.

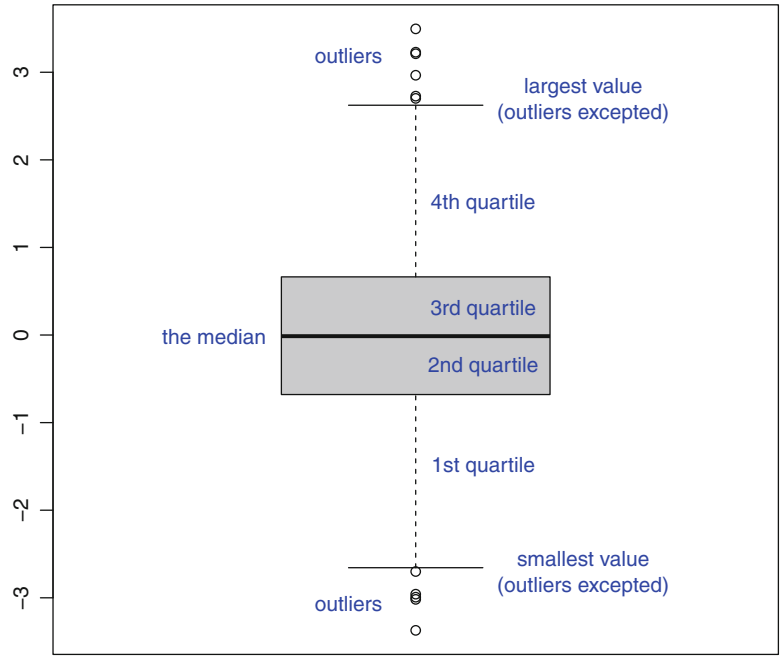


Fig. 7.4: A generic boxplot

You obtain Fig. 7.4 with the `boxplot()` function.

```
> boxplot(rnorm(1000), col="grey")
```

Note that the boxplot will look different each time you enter the code because `rnorm(1000)` generates 1000 random values of the normal distribution (enter `?Normal` for further information). The center of the plot consists of a box that corresponds to the middle half of the data. The height of the box is determined by the interquartile range. The thick horizontal line that splits the box in two is the median. If the median is centered between the lower limit of the second quartile and the upper limit of the third quartile, as is the case in Fig. 7.4, this is because the central part of the data is roughly symmetric. If not, this is because the frequency distribution is skewed. Whiskers are found on either side of the box. They go from the upper and lower limits of the box to the horizontal lines of the whiskers. These two lines are drawn 1.5 interquartile ranges above the third quartile or 1.5 interquartile ranges below the first quartile. If whiskers have different lengths, it is also a sign that the frequency distribution is skewed. The data values beyond the whiskers are known as outliers. They are displayed individually as circle dots. Although connoted negatively, outliers can be interesting and should not be systematically ignored.

Regarding the `split_unsplit.rds` data set, we use boxplots to compare the dispersion of the frequency distributions of split and unsplit infinitives. There are two ways of doing it. The first way consists in selecting the desired columns of the data frame (Fig. 7.5a). Here, this is done via subsetting.

```
> boxplot(data[,c(2,3)])
```

The second way consists in plotting each variable side by side as two separate vectors (Fig. 7.5b). The variables are not labeled.

```
> boxplot(data$UNSPLIT_INFINITIVE, data$SPLIT_INFINITIVE)
```

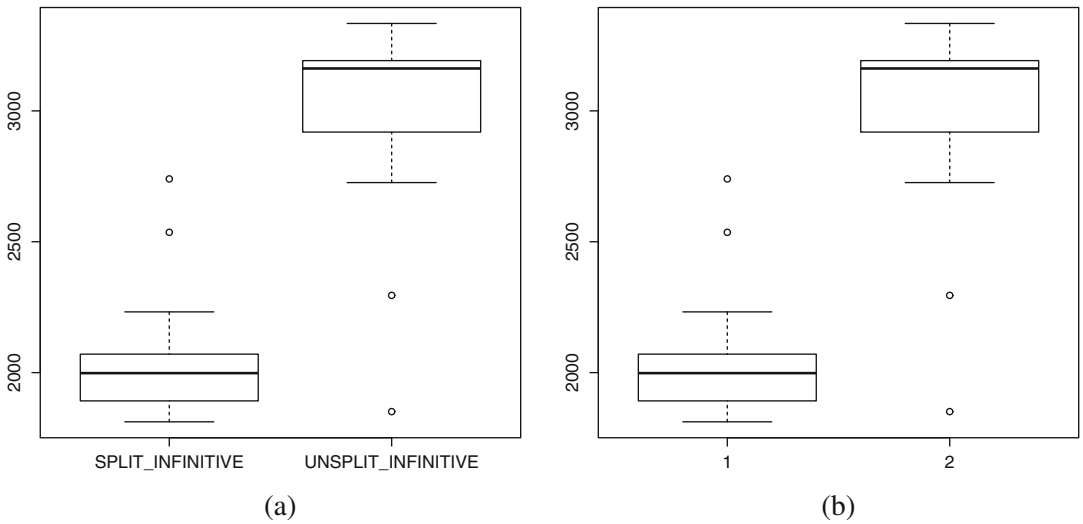


Fig. 7.5: Boxplots for `SPLIT_INFINITIVE` and `UNSPLIT_INFINITIVE`. (**a**) Plotting columns; (**b**) plotting vectors

If we were to superimpose these two boxplots, they would not overlap. This shows that their distributions are radically different. The boxplot for `SPLIT_INFINITIVE` is shorter than the boxplot for `UNSPLIT_INFINITIVE` because the frequency distribution of the former variable is less dispersed than the frequency distribution of the latter. The boxplot for `UNSPLIT_INFINITIVE` shows that the frequency distribution for this variable is skewed: the whiskers do not have the same length, and the median is close to the upper limit of the third quartile.

7.3.3 Variance and Standard Deviation

The variance and the standard deviation use the mean as their central point. The variance (σ^2) is calculated by (a) subtracting the mean ($\bar{x}$) from each data point (x), (b) squaring the difference, (c) summing up all squared differences, and (d) dividing the sum by the sample size (N) minus 1.

$$\sigma^2 = \frac{\sum (x - \bar{x})^2}{N - 1} \tag{7.2}$$

Fortunately, R has a built-in function for the variance: `var()`.

```
> var(data$SPLIT_INFINITIVE)
[1] 53799.94
> var(data$UNSPLIT_INFINITIVE)
[1] 132495.1
```

The standard deviation (σ) is the most widely used measure of dispersion. It is the square root of the variance.

$$\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{N - 1}} \tag{7.3}$$

In R, you obtain the standard deviation of a frequency distribution either by first calculating the variance of the vector and then its square root

```
> sqrt(var(data$SPLIT_INFINITIVE))
[1] 231.9481
> sqrt(var(data$UNSPLIT_INFINITIVE))
[1] 363.9987
```

or by applying the dedicated function: `sd()`.

```
> sd(data$SPLIT_INFINITIVE)
[1] 231.9481
> sd(data$UNSPLIT_INFINITIVE)
[1] 363.9987
```

As expected, the variance and the standard deviation of `UNSPLIT_INFINITIVE` are larger than the variance and the standard deviation of `SPLIT_INFINITIVE`.

Exercises

7.1. Central tendency

The data file for this exercise is `modals.by.genre.BNC.rds`, to be found in `CLSR/chap6`. Write a `for` loop that prints (with `cat()`):

- the mean number of modals per text genre;
- the median number of modals per text genre;
- the frequency of the mode, i.e. the modal that occurs the most in the given text genre.

The `for` loop should output something like the following:

```
ACPROSE 20940.1 18399.5 44793
CONVRSN 6361.8 3934 23161
FICTION 21789.5 14480 56934
NEWS 11476.8 7517 37476
NONAC 29072.6 29272.5 59211
OTHERPUB 23729.9 20303.5 66980
OTHERSP 9967.2 6178.5 26262
UNPUB 6684.5 6293 19258
```

7.2. Valley speak

A sociolinguist studies the use of the gap-filler 'whatever' in Valleyspeak. She records all conversations involving two Valley girls over a week and counts the number of times each girl says 'whatever' to fill a gap in the conversation. The (fictitious) data is summarized in Tab. 7.1.

Table 7.1: Data on Valleyspeak

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Valley girl 1	314	299	401	375	510	660	202
Valley girl 2	304	359	357	342	320	402	285

For each girl:

1. summarize the data in a plot such as Fig. 7.6;
2. compute the mean and the standard deviation;
3. summarize the data in a boxplot;
4. interpret the results.

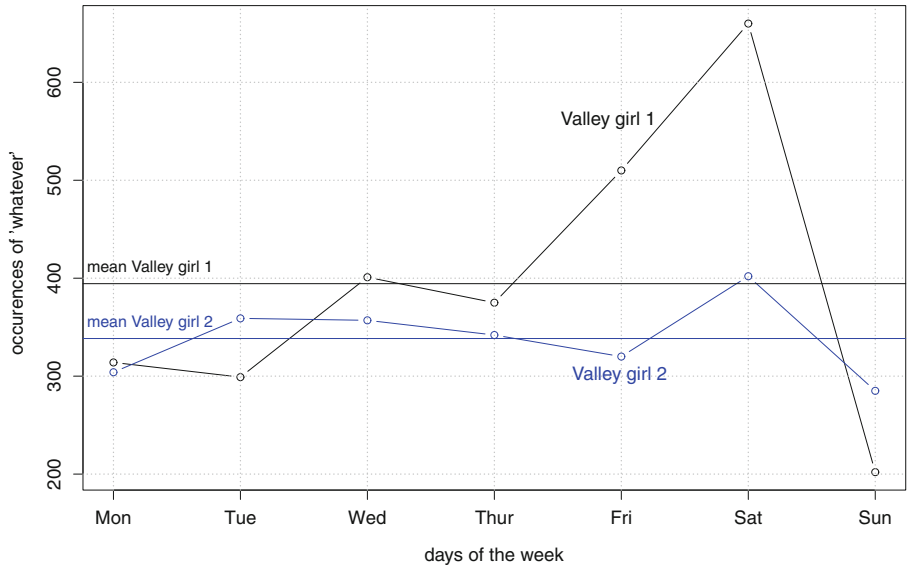


Fig. 7.6: Number of times two Valley girls say 'whatever' over a week

Chapter 8
Notions of Statistical Testing

Abstract In this chapter, you will learn the basics of statistical thinking, namely inferential statistics and statistical testing. These fundamentals will serve as a basis for the following chapters.

8.1 Introduction

The previous chapter introduced descriptive statistics, which are a prerequisite to the kinds of statistics introduced in the present chapter, namely inferential statistics and statistical testing. Inferential statistics consists in generalizing what happens in a sample to the whole population of your data. Before we proceed further, let us review the key concepts implied in statistical thinking. These concepts will be used from this chapter onwards.

8.2 Probabilities

When you say that an event is unlikely, likely, or certain to happen, you make a probability judgment. English has lots of expressions related to probability judgments: "what are the odds?", "odds/chances are that…", etc. In this section, I will show you the relevance of probabilities in the quantitative study of language.

8.2.1 Definition

In baseball, if you bet that the Oakland A's will beat the Texas Rangers, it is because you consider this outcome (an Oakland A's win) to be more likely than the others (an Oakland A's defeat or a draw). A probability is expressed on a continuous scale from 0 to 1: 0 means that the outcome has no chance to occur, 0.5 that it has one chance out of two to occur, and 1 that it is certain to occur. You may also express a probability by means of a percentage: 0%, 50%, and 100% mean that the outcome has no chance to occur, one chance out of two to occur, or is certain to occur respectively. Let X be the event that the Oakland A's will win. If you consider that the A's have three chances out of four to win, you express this probability as

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_8

follows: $P(X) = 0.75$. Suppose that a draw is impossible because this is the last game in the World Series. You know that the outcome 'win' and the outcome 'lose' are mutually exclusive. The probability that the A's will lose ($\bar{X}$) is therefore:

$$P(\bar{X}) = 1 - 0.75 = 0.25. \tag{8.1}$$

If you bet that the Oakland A's will win, you certainly assume that $P(\bar{X}) < 0.5 < P(X)$. If a friend bets that the Texas Rangers will win, she certainly assumes that $P(\bar{X}) > 0.5 > P(X)$. The only way to know for sure which is the winning bet is to wait until the baseball game has ended.

8.2.2 Simple Probabilities

Let us turn to a linguistic example. We draw uniformly (i.e. completely at random) a word from one BNC Baby file, namely `A1J.xml`. Let A be the event "the word drawn from the corpus file is a noun", and $P(A)$ the probability of that event. We want to calculate $P(A)$.

The code below is used to make a frequency list of the tags contained in the corpus file.

```
> rm(list=ls(all=TRUE))
> library(gsubfn)
> # load the corpus file
> corpus.file <- scan("/CLSR/BNC_baby/A1J.xml", what="char", sep="\n")
> # select the sentences in the corpus file
> sentences <- grep("<s n=", corpus.file, value=TRUE)
> # look for the tags
> tags <- unlist(strapplyc(sentences, "w c5=\"(.*?)\""))
> # tabulate and sort
> table_tags <- sort(table(tags), decreasing=TRUE)
> # convert the array into a tab-delimited table
> freqlist_tags <- paste(names(table_tags), table_tags, sep="\t")
```

The table itself is available from your `CLSR` folder.

```
> freqlist <- readRDS("C:/CLSR/chap8/freqlist.A1J.rds") # Windows
> freqlist <- readRDS("/CLSR/chap8/freqlist.A1J.rds") # Mac
```

Thanks to the frequency list, you know that there are 13663 words in the corpus file.

```
> W <- sum(freqlist$FREQUENCY); W
[1] 13663
```

According to the BNC tagset, nouns are coded `NN0`, `NN1`, `NN2`, or `NP0` (see Sects. 5.3.2 and 6.4). Thanks to this information, you can calculate the total number of nouns by subsetting the data frame and apply `sum()` to `FREQUENCY`.

```
> nouns <- c("NN0", "NN1", "NN2", "NP0")
> freqlist.nouns <- freqlist[freqlist$TAG %in% nouns, ]
> NP <- sum(freqlist.nouns$FREQUENCY); NP
[1] 4047
```

There are 4047 nouns in `A1J.xml`. The probability of drawing a noun from `A1J.xml` is $P(A) = \frac{\texttt{NP}}{\texttt{W}} = \frac{4047}{13663} = 0.296$.

```
> P_A <- NP/W; P_A
[1] 0.2962014
```

In other words, there is a 29.6% chance of drawing a noun from the corpus file. The probability of drawing a word that is not a noun is $P(\bar{A}) = 1 - P(A) = 0.704$ (70.4%).

```
> 1-P_A
[1] 0.7037986
```

A more complex, though more instructive, way to obtain the probability of drawing a noun is to calculate the probability to draw each type of noun separately from the corpus file.

```
> # subset the data frame
> freqlist.NN0 <- freqlist[freqlist$TAG %in% "NN0", ]
> freqlist.NN1 <- freqlist[freqlist$TAG %in% "NN1", ]
> freqlist.NN2 <- freqlist[freqlist$TAG %in% "NN2", ]
> freqlist.NP0 <- freqlist[freqlist$TAG %in% "NP0", ]
>
> # calculate the probability for each type of noun
> P_NN0 <- sum(freqlist.NN0$FREQUENCY)/W
> P_NN1 <- sum(freqlist.NN1$FREQUENCY)/W
> P_NN2 <- sum(freqlist.NN2$FREQUENCY)/W
> P_NP0 <- sum(freqlist.NP0$FREQUENCY)/W
```

Now, all you need is to sum up these probabilities.

```
> P_NN0 + P_NN1 + P_NN2 + P_NP0
[1] 0.2962014
```

As expected, you get the same result. This is known as the additive principle. It is at work when the variables are mutually exclusive (hence not independent). If you draw a common noun, neutral for number such as *aircraft* (`NN0`), you cannot draw a singular common noun, such as *pencil*. If we sum up the probabilities of all tags in the data frame, we obtain 1.

8.2.3 Joint and Marginal Probabilities

Now, suppose that three corpus files are involved instead of just one: `A1J.xml`, `A1K.xml`, and `A1L.xml`. Tab. 8.1 groups frequency counts from two variables: the type of noun (rows) and the corpus file (columns). This two-way table is an example of a contingency table.

Table 8.1: The distribution of nouns in three BNC Baby corpus files

	`A1J.xml`	`A1K.xml`	`A1L.xml`	row totals
`NN0`	136	14	8	158
`NN1`	2236	354	263	2853
`NN2`	952	87	139	1178
`NP0`	723	117	71	911
column totals	4047	572	481	5100

The row totals reveal that there are 158 common nouns, neutral for number, 2853 singular common nouns, 1178 plural common nouns, and 911 proper nouns in the three corpus files. The column totals reveal that

there are 4047 nouns in `A1J.xml`, 572 nouns in `A1K.xml`, and 481 nouns in `A1L.xml`. The number 5100 in the lower right corner gives the total number of nouns in the three corpus files, regardless of the type. This total is found by summing up the row totals, the column totals, or the 12 cells of the contingency table.

Tab. 8.1 is available in your `/CLSR/chap8` subfolder.[1]

```
> cont.table <- readRDS("C:/CLSR/chap8/cont.table.rds"); cont.table # Windows
> cont.table <- readRDS("/CLSR/chap8/cont.table.rds"); cont.table # Mac
```

```
       A1J.xml A1K.xml A1L.xml Total
NN0        136      14       8   158
NN1       2236     354     263  2853
NN2        952      87     139  1178
NP0        723     117      71   911
Total     4047     572     481  5100
```

Let *NN1* be the event: "the word that is drawn is a singular common noun" and *A1K* the event "the word that is drawn is from `A1K.xml`". The event that the word that is drawn is a singular common noun from `A1K.xml` is a joint event. Let *NNO&A1K* denote this joint event for now. It is realized 354 times in Tab. 8.1, which contains 12 such joint events, all of which are mutually exclusive.

The total number of nouns in all three corpus files (N) allows us to calculate the probabilities of the events *NN1* and *A1K*.

```
> N <- sum(cont.table[1:4, 1:3])
> P_NN1 <- cont.table[2,4]/N; P_NN1
[1] 0.5594118
> P_A1K <- cont.table[5,2]/N; P_A1K
[1] 0.1121569
```

These are known as marginal probabilities because they are calculated on the basis of events found in the margins of the contingency table. N also allows us to calculate joint probabilities, such as $P(NN1\,\&\,A1K)$, the probability of drawing a word that is a singular common noun from `A1K`.

```
> P_NN1_and_A1K <- 354/N; P_NN1_and_A1K
[1] 0.06941176
```

$P(NN1\,\&\,A1K) \approx 0.069$, which means that there is a 6.9% chance of drawing a singular common noun from corpus file `A1K.xml`. All the joint probabilities in the table can be calculated in one go by dividing the whole table by N.

```
> probs <- cont.table/N; probs
            A1J.xml     A1K.xml     A1L.xml      Total
NN0   0.02666667 0.002745098 0.001568627 0.03098039
NN1   0.43843137 0.069411765 0.051568627 0.55941176
NN2   0.18666667 0.017058824 0.027254902 0.23098039
NP0   0.14176471 0.022941176 0.013921569 0.17862745
Total 0.79352941 0.112156863 0.094313725 1.00000000
```

Tab. 8.1 displays a joint frequency distribution, whereas the table stored in `probs` displays a joint probability distribution.

[1] The code to generate the table is available from Sect. A.2.1.

8.2.4 Union vs. Intersection

When we are interested in the probability of drawing a word that is *both* a singular common noun *and* from `A1K`, we are talking about the *intersection* of two events. That which we expressed above by means of the symbol "&" is in fact denoted by means of the $\cap$ symbol from set theory. This probability is conventionally written in the form: $P(NN1 \cap A1K)$.

Coming back to Tab. 8.1, suppose that we are now interested in the probability of drawing *either* a singular common noun *or* a noun from `A1K`. This is an illustration of the *union* of two events, which is denoted by means of the $\cup$ symbol from set theory. This probability is written in the form: $P(NN1 \cup A1K)$. It is calculated as follows:

$$P(NN1 \cup A1K) = P(NN1) + P(A1K) - P(NN1 \cap A1K)$$

We have all the information that we need to calculate this probability in R.

```
> P_NN1_or_A1K <- P_NN1 + P_A1K - P_NN1_and_A1K ; P_NN1_or_A1K
[1] 0.6021569
```

$P(NN1 \cup A1K) = 0.602$ (60.2%).

8.2.5 Conditional Probabilities

Computing the probability of a joint event is not the same as computing the probability of an event assuming that another event occurs. The latter is known as a conditional probability. For example, we might want to know the probability that a word is a singular common noun given that `A1K.xml` is selected. This conditional probability is denoted $P(NN1|A1K)$. Using Tab. 8.1, we divide the frequency of singular common nouns in `A1K.xml` (354) by the total of column `A1K` (572).

```
> P_NN1_given_A1K <- cont.table[2,2]/cont.table[5,2]; P_NN1_given_A1K
[1] 0.6188811
```

When `A1K.xml` is selected, there is a 61.89% chance of drawing a singular common noun. There is a formula to compute conditional probabilities. Let A and B be any two events with $P(A) > 0$, then:

$$P(A|B) = \frac{P(A \cap B)}{P(A)}. \tag{8.2}$$

Therefore,

$$P(NN1|A1K) = \frac{P(NN1 \cap A1K)}{P(A1K)}. \tag{8.3}$$

In R, we compute $P(NN1|A1K)$ thanks to the information we already have.

```
> P_NN1_and_A1K/P_A1K
[1] 0.6188811
```

Finally, let us compare $P(NN1|A1K)$ to $P(A1K|NN1)$, i.e. the probability that the word is drawn from `A1K.xml` given that the word is a noun.

$$P(A1K|NN1) = \frac{P(NN1 \cap A1K)}{P(NN1)} \tag{8.4}$$

```
> P_A1K_given_NN1 <- P_NN1_and_A1K/P_NN1; P_A1K_given_NN1
[1] 0.1240799
```

When a singular common noun is selected, there is a 12.4% chance that this word is drawn from `A1K.xml`.

8.2.6 Independence

Two events can be statistically dependent or independent. If two events are dependent, the occurrence of one event affects the probability of the other event. If two events are independent, the occurrence of one event does not affect the probability of the other event.

As a rule, you prove that two events A and B are independent if

$$P(B|A) = P(B) \tag{8.5}$$

or if

$$P(A|B) = P(A). \tag{8.6}$$

Let us apply the first formula to see if

$$P(NN1|A1K) = P(NN1) \tag{8.7}$$

and decide if *NN1* and *A1K* are independent.

```
> P_NN1_given_A1K
[1] 0.6188811
> P_NN1
[1] 0.5594118
```

Event *NN1* is not independent of event *A1K* because the probability of *NN1* given *A1K* (0.618) is not the same as the probability of *NN1* among all the corpus files (0.559). Events *NN1* and *A1K* are therefore dependent. When two events A and B are dependent, another rule applies:

$$P(A\,\&\,B) = P(A) \times P(B|A). \tag{8.8}$$

This rule is verified with respect to *NN1* and *A1K*:

$$P(NN1\,\&\,A1K) = P(NN1) \times P(A1K|NN1). \tag{8.9}$$

```
> P_NN1_and_A1K
[1] 0.06941176
> P_NN1*P_A1K_given_NN1
[1] 0.06941176
> # alternatively, use the function identical()
> identical(P_NN1_and_A1K, P_NN1*P_A1K_given_NN1)
[1] TRUE
```

Suppose now that instead of drawing one word uniformly, we successively draw two words uniformly with replacement regardless of the corpus file. "With replacement" means that the word drawn first is not removed from the total sample of nouns before drawing the second word. Let *NN1* be the event "the (first or second) word is a singular common noun" and *NP0* the event "the (first or second) word is a proper noun". Given the above rule (8.8), *NN1* and *NP0* are independent if

$$P(NN1|NP0) = P(NN1), \tag{8.10}$$

or if

$$P(NP0|NN1) = P(NP0). \tag{8.11}$$

You do not need to make any calculation. Drawing one type of noun does not affect the probability of drawing another type of noun, therefore both events are independent. The probability of an event given the other is therefore the same as the probability of the former event. Things would be different if the word was removed from the total sample.

When two events A and B are independent, the multiplication principle applies:

$$P(A\,\&\,B) = P(A) \times P(B). \tag{8.12}$$

The probability of drawing a singular common noun and a plural common noun is calculated as follows:

$$P(NN1\,\&\,NP0) = P(NN1) \times P(NP0). \tag{8.13}$$

```
> P_NP0 <- probs[4,4]
> P_NN1_and_NP0 <- P_NN1*P_NP0; P_NN1_and_NP0
[1] 0.0999263
```

The probability of drawing a singular common noun and a plural common noun is approximately 0.1 (10%).

8.3 Populations, Samples, and Individuals

Suppose you investigate a linguistic phenomenon such as cleft sentences in English. An ideal experiment would consist in using descriptive statistics to summarize the distribution of all the cleft sentences found in an immense library gathering all spoken and written English documents produced by native English speakers during a period of interest. Of course, this kind of study is not feasible because the immense library does not exist. If it were possible, perhaps it would not be practical because of the amount of observations involved.

What corpus linguists do instead is a less ideal but more practical experiment that consists in drawing conclusions about the language based on a sample of that language. This method is inferential. In statistics parlance, the collection of all possible observations in the study is known as the population. The population may be finite or potentially infinite. The part of the population from which information is actually obtained is called a sample. In Fig. 8.1, all the dots represent the population. The red dots represent the sample drawn from the population.

These concepts are relative. For example, if you investigate a subcorpus hoping to make an inference on the whole corpus that it belongs to based on information from the subcorpus, the subcorpus is your sample

and the whole corpus your population. Following the same demographic metaphor, each observation of the case study is an individual. In Sect. 1.2.1.1, we saw that a corpus could also be considered a sample of samples, a sample drawn from the population of utterances in a given language. In this sense, a reference corpus such as the British National Corpus could be considered a sample to make inferences about the population, namely British English.

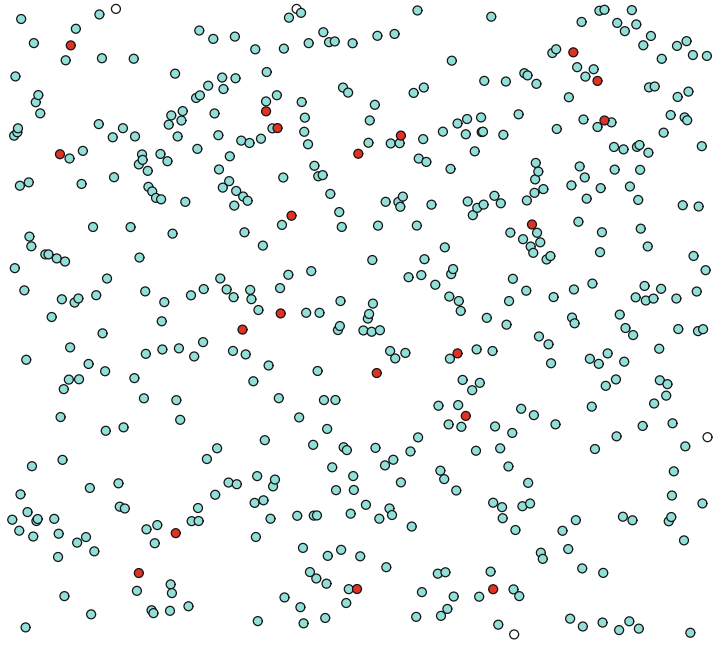


Fig. 8.1: A population and a sample

8.4 Random Variables

In Sect. 7.1, we defined the variable. At this stage, let me refine the definition a little by introducing a central (and perhaps foundational) concept in statistics: randomness. A random phenomenon occurs by chance. A random variable is like a variable except that the values that it may take depend on chance. Mathematicians feel comfortable with random phenomena because it is something that they can model.

If you have read books on statistics, you must have realized that statisticians love tossing coins or throwing dice. My own preference goes to dice rolling for an etymological reason. "Hasard" is the French word for "chance". It derives from "al-zahr", which is Arabic for "dice". Here is a classical example. You play a gambling game with a friend: each of you rolls a die 100 times in a row. If the outcome is an even number (⚁, ⚂, or ⚃), you win. If the outcome is an odd number (⚀, ⚁, or ⚂), you lose. Each roll is a *trial*. Trials are *independent* from each other (i.e. the outcome of a trial does not affect the outcome of another event). The outcome of the roll of the die is either an even number or an odd number. The probability p for either event is: $p_{even} = p_{odd} = 3/6 = 1/2 = 0.5$.

Linguists, however, hardly ever see language phenomena as random. The lines below are from Kilgarriff (2005, p. 264), a landmark paper I believe all linguists should read before embarking on quantitative corpus linguistics.

> The problem for empirical linguistics is that language is not random (. . .). Language is not random because we speak or write with purposes. We do not, indeed, without computational help are not capable of, producing words or sounds or sentences or documents randomly.

Like Kilgarriff, not all linguists find drawing linguistic units at random from a sample natural. Some even find this practice irrelevant. They will rightly argue that a linguistic sample is not just a bag of words and a sentence is not the random sum of the N words it is made of. The order of linguistic units is primordial, and linguistics is precisely the study of the constraints that govern this order. However, statistics tells you that by comparing what happens in your sample to what a theoretical model based on probabilities posits is what allows you to measure precisely the effect of linguistic constraints at work in the sample. For this reason, quantitative linguists claim that, even though inferential statistics relies largely on the use probability models whose randomness assumptions are never met in language, these models are in fact useful, providing that you take their limitations into account.

8.5 Response/Dependent vs. Explanatory/Descriptive/Independent Variables

As seen in Sect. 1.2.3.1, you do not collect corpus data at random. You do it because you are interested in a linguistic phenomenon. More precisely, your prime objective is to investigate the effects of descriptive variables on the variable of interest. By convention, this variable of interest, whose investigation is the true goal of your study, is called the response variable or the dependent variable. A variable used to describe the response/dependent variable is called a descriptive variable, an independent variable, or an explanatory variable.

How your data are organized in a data frame should be planned before the corpus extraction (see Sect. 5.3). As a rule, and also to facilitate further processing in R, each observation should have its own row. In the context of corpus linguistics, an observation corresponds to a match. Each variable (response and descriptive) should have its own column. Tab. 8.2 is what you should aim at. This format allows you to further annotate each observation easily (the column "semantic class" was annotated manually).

Table 8.2: A conventional data frame (excerpt)

case	corpus file	construction	intensifier	text mode	text type	text info	match	adjective	semantic class
1	KBP.xml	PREDETERMINER	QUITE	SPOKEN	CONVRSN	S conv	quite an able chap	able	Able/intelligent
2	H9C.xml	PREADJECTIVAL	RATHER	WRITTEN	FICTION	W fict prose	a rather clever pupil	clever	Able/intelligent
3	A7Y.xml	PREDETERMINER	QUITE	WRITTEN	NEWS	W newsp brdsht nat: social	quite a clever move	clever	Able/intelligent
4	FAJ.xml	PREDETERMINER	QUITE	WRITTEN	FICTION	W fict prose	quite a deft touch	deft	Able/intelligent
5	HR9.xml	PREDETERMINER	RATHER	WRITTEN	FICTION	W fict prose	rather a shrewd point	shrewd	Able/intelligent
6	AHC.xml	PREADJECTIVAL	QUITE	WRITTEN	NEWS	W newsp brdsht nat: misc	a quite different class	different	Comparison_Different
7	J10.xml	PREADJECTIVAL	QUITE	WRITTEN	FICTION	W fict prose	a quite different direction	different	Comparison_Different
8	J10.xml	PREADJECTIVAL	QUITE	WRITTEN	FICTION	W fict prose	a quite different tack	different	Comparison_Different
9	CTY.xml	PREADJECTIVAL	RATHER	WRITTEN	ACPROSE	W ac:humanities arts	a rather different critique	different	Comparison_Different
10	H9D.xml	PREADJECTIVAL	RATHER	WRITTEN	FICTION	W fict prose	a rather different kind	different	Comparison_Different
…	…	…	…	…	…	…	…	…	…

8.6 Hypotheses

Before you collect corpus data, you must ask a theory-informed question, which you convert into a research hypothesis. Basically, you make a statement about something that you suppose to be the case and then collect corpus evidence based on this statement.

The data frame overviewed in Tab. 8.2 is based on Desagulier (2015), a study of the pre-adjectival vs. pre-determiner alternation in the context of adjectival intensification with *quite* and *rather* in the BNC (we shall return to it in the next chapter). Previous literature suggests that the choice of pre-determiner position over the default pre-adjectival position depends on style or formality. This simple claim can be easily turned into a hypothesis. A hypothesis breaks down into two parts: the alternative hypothesis and the null hypothesis. The alternative hypothesis (H_1) posits a relation of dependence between the response variable and the explanatory variables. The null hypothesis (H_0) posits a relation of independence between them. If the syntax of the intensifier (preadjectival vs. predeterminer) is our response/dependent variable and stylistic/contextual features our explanatory/descriptive/independent variables, H_1 and H_0 may look like the following:

> H_1: The choice of pre-determiner position over the default pre-adjectival position depends on style or formality;
> H_0: The choice of pre-determiner position over the default pre-adjectival position does not depend on style or formality.

What remains to determine is how style and formality are operationalized. Operationalization is the conversion of your textual hypotheses into something that can be observed, measured, or counted. There are two choices to make. The first choice concerns how to convert abstract concepts such as "style or formality" into tangible variables. This is where your linguist's expertise comes in. For convenience sake, let us assume that written texts are more formal than spoken transcripts.[2]

The second choice has to do with quantification. The trick is to collect your data so that they are amenable to quantification (see the previous chapter) and statistics (see below). The data frame `quite.rather.rds` has been planned so that the variables "text mode" and "text type" capture style and formality. Also part of the original design of the data frame is the fact that the nominal variables can yield frequency counts. Let us see how this works.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap8/quite.rather.rds") # Windows
> data <- readRDS("/CLSR/chap8/quite.rather.rds") # Mac
```

Next, subset the data frame to select the desired variables and summarize the data frame with `summary()`.

```
> data <- data[,c(2:5)] # subset
> summary(data) # summarize the data frame
        construction intensifier    text_mode     text_type
 PREADJECTIVAL: 59   QUITE :103   SPOKEN : 57   ACPROSE:31
 PREDETERMINER:104   RATHER: 60   WRITTEN:106   CONVRSN:57
                                                FICTION:47
                                                NEWS   :28
```

As you can see, nominal data converted easily into frequency counts. Thanks to the `count()` function from the `plyr` package, you can turn a whole table of nominal data into a frequency table in just one line of code. The variables to be summarized should be combined with `c()` in the `vars` argument of the `count()` function.

[2] Admittedly, every linguist knows that some written texts (such as emails, text messages, or songs) can be far less formal than some spoken transcripts (like official speeches). Furthermore, there are several degrees of formality per mode.

```
> library(plyr)
> freq.table <- count(data, vars = c("construction", "intensifier", "text_mode", "text_type"))
> freq.table
    construction intensifier text_mode text_type freq
1  PREADJECTIVAL       QUITE    SPOKEN   CONVRSN    2
2  PREADJECTIVAL       QUITE   WRITTEN   ACPROSE    3
3  PREADJECTIVAL       QUITE   WRITTEN   FICTION    4
4  PREADJECTIVAL       QUITE   WRITTEN      NEWS    2
5  PREADJECTIVAL      RATHER   WRITTEN   ACPROSE   21
6  PREADJECTIVAL      RATHER   WRITTEN   FICTION   21
7  PREADJECTIVAL      RATHER   WRITTEN      NEWS    6
8  PREDETERMINER       QUITE    SPOKEN   CONVRSN   52
9  PREDETERMINER       QUITE   WRITTEN   ACPROSE    5
10 PREDETERMINER       QUITE   WRITTEN   FICTION   17
11 PREDETERMINER       QUITE   WRITTEN      NEWS   18
12 PREDETERMINER      RATHER    SPOKEN   CONVRSN    3
13 PREDETERMINER      RATHER   WRITTEN   ACPROSE    2
14 PREDETERMINER      RATHER   WRITTEN   FICTION    5
15 PREDETERMINER      RATHER   WRITTEN      NEWS    2
```

The most obvious form of quantification consists of frequency counts.[3]

There are several descriptive variables whose interaction might give you a headache at this stage.[4] To keep things simple, we restrict the study (and the data set) to one response variable (the choice of the preadjectival vs. predeterminer construction) and one descriptive variable (text mode). We want to know if the choice of the preadjectival vs. predeterminer construction depends on whether the mode is written or spoken. Through operationalization, H_1 and H_0 become:

H_1: predeterminer and preadjectival constructions are not equally distributed across text modes;
H_0: predeterminer and preadjectival constructions are equally distributed across text modes.

H_1 can be directional ("one-tailed") or non-directional ("two-tailed"). As formulated above, the alternative hypothesis is non-directional: although it postulates that the choice of the construction depends on text mode, it does not assume a direction. More specifically, the alternative hypothesis does not tell you that one construction is overrepresented in a text mode. Suppose that you have read somewhere that predeterminer constructions are restricted to written texts. If you want to test that assumption, you formulate a directional alternative hypothesis and the corresponding null hypothesis,[5] which may look like one of the following:

H_1: the predeterminer construction occurs more often in written contexts;
H_0: the predeterminer construction occurs less often in written contexts; or
H_0: the predeterminer construction occurs equally often in written and spoken contexts;

or

H_1: the predeterminer construction occurs less often in written contexts;
H_0: the predeterminer construction occurs more often in written contexts; or
H_0: the predeterminer construction occurs equally often in written and spoken contexts.

How you formulate the alternative hypothesis depends on what you want to show. What you want to show depends on your review of the literature on the topic and your qualitative assessment of the case study prior to corpus extraction.

Once you have determined H_1 and H_0 in textual form, you turn them into a statistically relevant form. Let f denote the frequency of a construction, and $\bar{f}$ the mean frequency. The alternative hypotheses become:

[3] Some tendencies emerge, such as the high frequencies of predeterminer *quite* in spoken conversation, and preadjectival *rather* in two written contexts: academic prose and fiction. The statistical significance of these tendencies remains to be shown.

[4] In Chap. 10, we see how to handle the simultaneous interaction of multiple variables with multivariate statistics.

[5] Often, there are two corresponding null hypotheses. Proving that one of them is false is enough to prove that H_1 is correct.

H_1 (directional): $\bar{f}_{predeterminer\_written} > \bar{f}_{predeterminer\_spoken}$;
H_0: $\bar{f}_{predeterminer\_written} \leqslant \bar{f}_{predeterminer\_spoken}$;

or

H_1 (directional): $\bar{f}_{predeterminer\_written} < \bar{f}_{predeterminer\_spoken}$;
H_0: $\bar{f}_{predeterminer\_written} \geqslant \bar{f}_{predeterminer\_spoken}$.

Your hypotheses can now be tested. Out of scientific integrity, you can change neither H_1 nor H_0 once you have selected a set of hypotheses and obtained results.

8.7 Hypothesis Testing

In hypothesis testing, you do not prove that your alternative hypothesis is true. You prove that the null hypothesis is false. This principle, known as hypothesis falsification, was laid out by Ronald Fisher in 1935:

> The null hypothesis is never proved or established, but is possibly disproved, in the course of experimentation. Every experiment may be said to exist only in order to give the facts a chance of disproving the null hypothesis. (Fisher 1935)

Once you have formulated H_1 and H_0 in textual and statistical forms, and before collecting your data, you define a threshold below which you will reject the null hypothesis. This threshold is known as the alpha level (α), the significance level, or the critical p-value. If you set α to 0.1, you consider that there is a 10% chance for the null hypothesis to be true. If you set it to 0.01, you consider that there is a 1% chance for the null hypothesis to be true. There is no real objective basis for setting α. Fisher (1935) himself admits that it is "situation dependent". If you are assessing the side effects of a medication, you will want to go with $\alpha = 0.01$ or lower. In linguistics and the humanities in general, where the stakes are not that high, the α level is generally set to 0.05, meaning that we can reject the null hypothesis at the 5% level of significance. Note that if your p-value is greater than α, this does not mean that the null hypothesis should be accepted (remember, you never prove that either H_1 or H_0 is true, you always reject its counterpart).

After defining the significance level, you compute the effect observed in your data using a statistic that is compatible with your operationalization (see previous section). This statistic should be paired with the probability of error p, which tells you how likely it is to find the effect observed in your data if H_0 is true. You can make two kinds of errors (Tab. 8.3). If you reject the null hypothesis when it is true, you are making a Type I error. This is known as a false positive. If you fail to reject the null hypothesis when it is false, you are making a type II error. This is known as a false negative.

Table 8.3: Type I and type II errors

	the truth	
your decision	H_0 is true	H_0 is false
you reject H_0	TYPE I error	no error
you fail to reject H_0	no error	Type II error

Finally, you come back to your hypothesis and make a decision. If $p < \alpha$, you reject H_0 and congratulate yourself. If $p > \alpha$ you reject H_1. Rejecting H_1 is not necessarily bad news. It can be quite meaningful with respect to the specificities of your data.

8.8 Probability Distributions

So far in this book we have worked with frequency distributions. The outcome of a counting experiment is always specific to the sample that it was computed from. If we replicate a word count on other samples, it is very likely that we will not get the same results. Tab. 8.4 shows the top ten words that you obtain when you make a frequency list of the eight text types in the BNC. Although we find roughly the same words at the top of each frequency list, the rankings and the frequency scores differ.

Table 8.4: Frequency lists for each of the 8 text types in the BNC

ACPROSE		FICTION		NEWS		NONAC	
word	frequency	word	frequency	word	frequency	word	frequency
the	2559	*the*	2349	*the*	254	*the*	315
of	1860	*of*	1183	*of*	129	*to*	208
a	1342	*to*	1170	*a*	116	*and*	201
in	1036	*and*	1160	*and*	104	*of*	198
to	918	*he*	1048	*in*	84	*a*	134
and	909	*wrote*	911	*to*	80	*in*	132
is	725	*I*	803	*is*	65	*for*	94
art	455	*it*	786	*s*	56	*with*	87
be	415	*in*	657	*that*	43	*is*	69
as	411	*is*	646	*with*	43	*I*	67
OTHERPUB		UNPUB		CONVRSN		OTHERSP	
word	frequency	word	frequency	word	frequency	word	frequency
the	156	*the*	1418	*the*	2004	*and*	306
to	115	*and*	763	*I*	1353	*the*	267
and	109	*of*	734	*and*	1231	*of*	189
of	95	*to*	676	*to*	1203	*to*	185
in	94	*in*	522	*that*	1099	*a*	139
a	59	*a*	480	*of*	1038	*in*	125
with	53	*for*	368	*you*	920	*was*	121
aids	48	*is*	314	*a*	893	*were*	103
is	43	*at*	191	*it*	860	*that*	102
are	41	*on*	186	*in*	760	*you*	97

However, many of the things that we study and measure are assumed to come from a predictable population of values that are distributed a certain way. If we plot the eight abovementioned frequency lists, we see that all the curves have the same shape (Fig. 8.2). It is the effect of the Zipfian distribution, alluded to in Sect. 6.2. The Zipfian distribution is an instance of the power law distribution. This distribution is characterized by a continuously decreasing curve, implying that a large number of rare events coexist with a handful

of large events. If you plot frequency lists based on samples of naturally occurring languages that are large enough, you will obtain curves that are shaped likewise. Thanks to the predictability of known distributions, we can calculate the probabilities that certain events or combinations of events will occur.

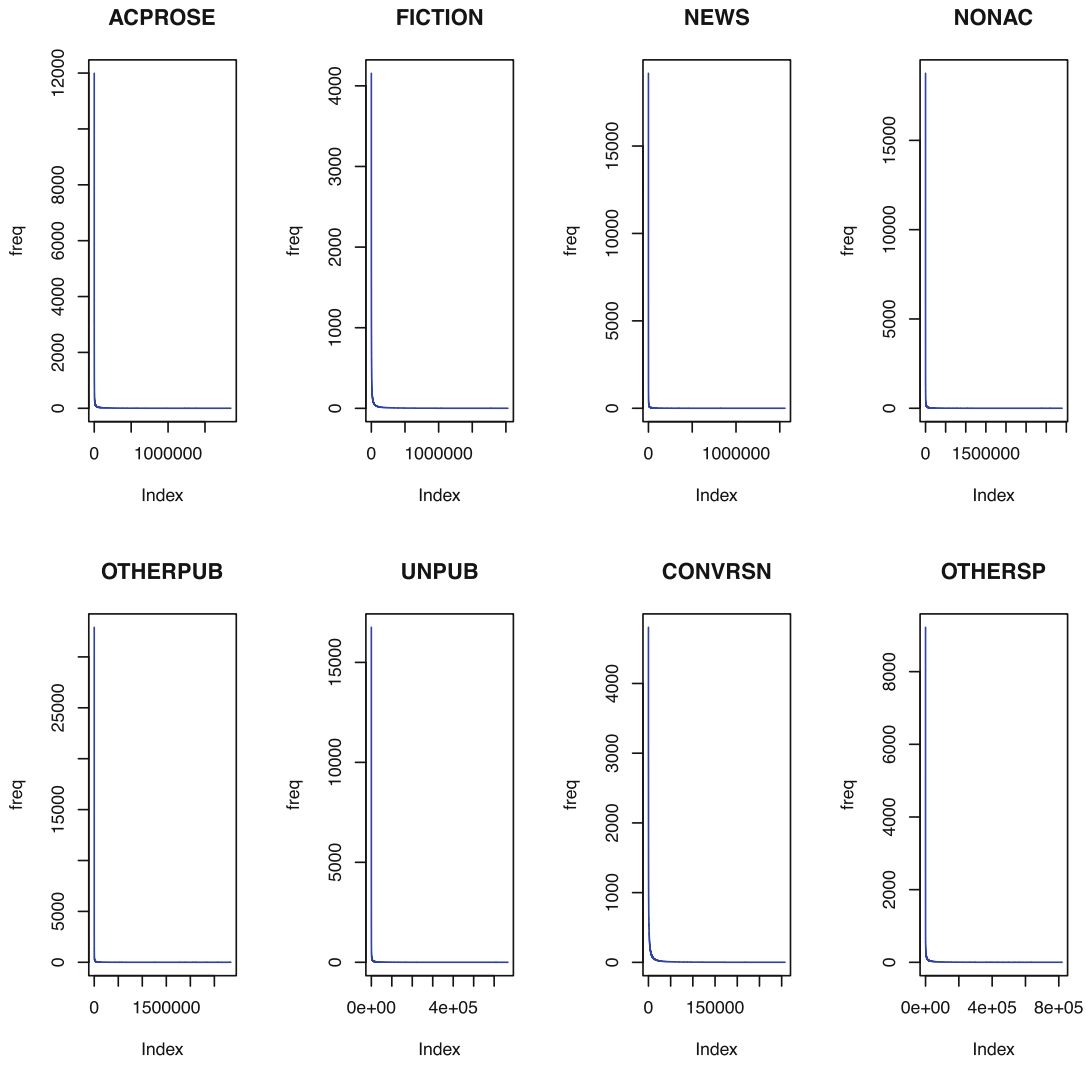


Fig. 8.2: Word distribution for each of the eight text types in the BNC

Like random variables, distributions subdivide into discrete and continuous distributions. The probability distribution of a discrete variable (such as word counts) can be used to indicate the probability of the outcome on an interval/ratio scale. Likewise, the probability distribution of a continuous variable (such as vowel length in millisecond, pitch in hertz) can be used to indicate the probability of the outcome on a continuous scale.

8.8.1 Discrete Distributions

In Sect. 8.4, I mentioned the classical experiment that involves two players who roll a die a hundred times in a row. If the outcome is an even number, one player wins. If the outcome is an odd number, the other player wins. We saw that each time, the probability for either event is $p = 0.5$. In other words, each outcome has one chance out of two to occur. The discrete probability distribution of the number of outcomes in a sequence of n dichotomous experiments is binomial.

Let me illustrate the binomial distribution with a linguistic example. You want to study the respective distributions of vowel and consonant letters in a text. You naively assume that vowels and consonants are used independently and randomly in a collection of poems. Each type of letter has one chance out of two to occur. We want to check if vowels are overrepresented. We set α to 0.05. We formulate hypotheses in textual form,

H_1: vowels are overrepresented,
H_0: vowels are not overrepresented.

Vowel and consonant letters lend themselves to counting. Taking into account the selected operationalization, the hypotheses become:

H_1: there are more vowels than expected by chance,
H_0: there are no more vowels than expected by chance.

Finally, we formulate the statistical hypotheses:

H_1: $p_V > 0.5$,
H_0: $p_V = p_C = 0.5$.

H_1 is directional because you assume that the effect goes in one direction (vowels are overrepresented). In statistical terms, you are looking for one-tailed p-values from discrete probability distributions. This will become clearer below. Now that you have formulated your hypotheses, you need to know how many times you draw vowels in a word before you can confidently say that they are overrepresented, i.e. before you can confidently reject H_0 and accept H_1.

Suppose you draw a four-letter word (not the one you are thinking about). There are 16 possible outcomes, which are summarized in Tab. 8.5. Because the events are independent, the multiplication principle applies and each has a probability of $0.5 \times 0.5 \times 0.5 \times 0.5 = 0.625$ (see Eq. 8.8). Under the naive assumption, vowels have no reason to occur less or more than twice in a four-letter word. Suppose that we find four vowels in the four-letter word, i.e. twice as more as expected. The probability associated with this outcome is $p_V = 0.625$, as the last line of the table indicates (in grey). Since $p_V > \alpha$, we reject H_1: finding four vowels in a four-letter word does not mean that the vowels are overrepresented. However, we should refrain from making definitive conclusions based on such a small number of trials. As you demultiply the number of trials, p_V drops dramatically. Compare the probability of finding a vowel all the time when we have 4, 10, 20, 30, 50, and 100 trials (Fig. 8.3).[6,7]

The shape of the curve is symmetric. We are only considering one portion of the curve (the "tail" to the right) because H_1 is directional. The properties of the binomial distribution are well known, and make it possible to estimate how much variability we may expect for the frequencies of vowels and consonants, given

[6] Admittedly, finding a 30-, 50- or 100-letter word is far from likely. As we will see below, the distribution of words based on the number of letters that they consist of is not a negligible fact.

[7] The code for generating Fig. 8.3 is available in Sect. A.2.2.

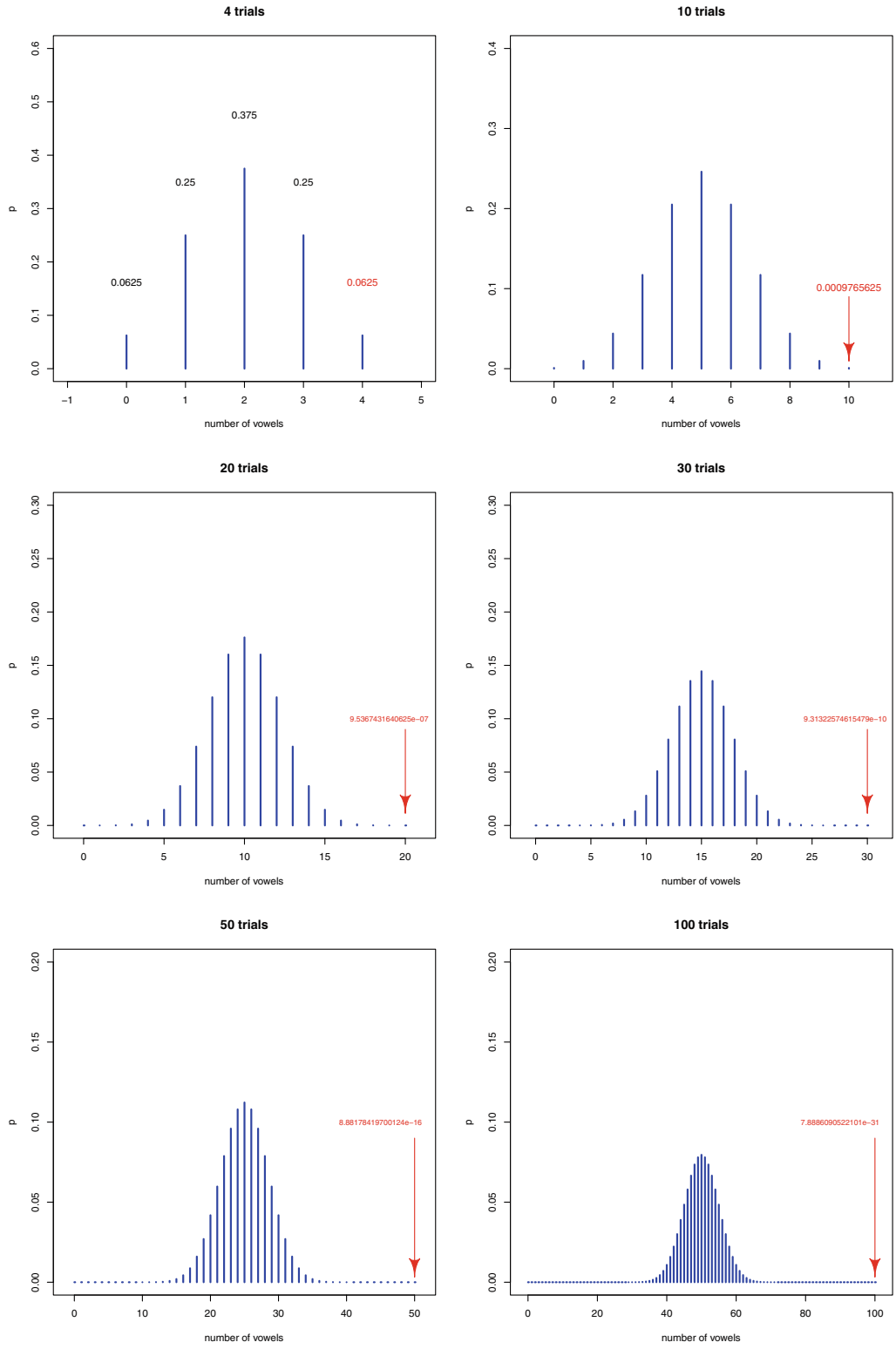


Fig. 8.3: Binomial probability distribution

Table 8.5: Possible outcomes

letter 1	letter 2	letter 3	letter 4	vowel count	consonant count	p
C	C	C	C	0	4	0.0625
C	C	C	V	1	3	0.0625
C	C	V	C	1	3	0.0625
C	C	V	V	2	2	0.0625
C	V	C	C	1	3	0.0625
C	V	C	V	2	2	0.0625
C	V	V	C	2	2	0.0625
C	V	V	V	3	1	0.0625
V	C	C	C	1	3	0.0625
V	C	C	V	2	2	0.0625
V	C	V	C	2	2	0.0625
V	C	V	V	3	1	0.0625
V	V	C	C	2	2	0.0625
V	V	C	V	3	1	0.0625
V	V	V	C	3	1	0.0625
V	V	V	V	4	0	0.0625

the naive assumption. In R, you obtain the probability of the first row in Tab. 8.5 thanks to the `dbinom()` function.[8]

```
> dbinom(4, size=4, prob=0.5)
[1] 0.0625
```

The first argument is the number of successes (finding four vowels), `size` is the number of trials, and `prob` is the probability of success on each trial (each letter type has a 0.5 chance of success). The probability of the first row in Tab. 8.5 is obtained likewise.

```
> dbinom(0, size=4, prob=0.5)
[1] 0.0625
```

If you want to know the cumulative probability of finding two, three, or four vowels in a four-letter word, you sum the respective probabilities of all trials.

```
> sum(dbinom(2:4, 4, 0.5))
[1] 0.6875
```

To understand better how this works, compute the cumulative probability of finding zero, one, two, three, or four vowels in a four-letter word. Because all possible outcomes are covered, you obtain 1. This corresponds to the total of the p column of Tab. 8.5.

```
> sum(dbinom(0:4, 4, 0.5))
[1] 1
```

Suppose that you draw 50 letters. How many vowels can you draw until you decide that they are overrepresented in the text? You can answer this question heuristically. Vowels are overrepresented if they appear

[8] In `dbinom`, `d` stands for "density". As far as concrete distributions are concerned, the density of x is the probability of getting exactly the value x.

more than 25 times. Let us calculate the cumulative probability of drawing 26 vowels or more out of 50 draws.

```
> sum(dbinom(26:50, 50, 0.5))
[1] 0.4438624
```

The p-value is larger than α (0.05) and does not allow us to accept H_1. This means that if you draw 26 vowels, it does not mean that vowels are overrepresented. We need to raise the number of times we draw a vowel until we find a cumulative probability that yields a p-value that is below 0.05.

```
> sum(dbinom(27:50, 50, 0.5))
[1] 0.3359055
> sum(dbinom(28:50, 50, 0.5))
[1] 0.2399438
> sum(dbinom(29:50, 50, 0.5))
[1] 0.1611182
> sum(dbinom(30:50, 50, 0.5))
[1] 0.1013194
> sum(dbinom(31:50, 50, 0.5))
[1] 0.05946023
> sum(dbinom(32:50, 50, 0.5))
[1] 0.03245432
```

Only when we draw 31 or more vowels out of 50 draws can we reject H_0 with a 5% level of significance and conclude that vowels are represented. R offers a non-heuristic way of getting this threshold number thanks to the `qbinom()` function.[9]

```
> qbinom(0.05, 50, 0.5, lower.tail = FALSE)
[1] 31
```

The first argument is α, the second argument is the number of trials, and the third argument is the probability of success on each trial. The fourth argument `lower.tail` is set to `FALSE` because H_1 is directional, i.e. one tailed.

Suppose now that we want to check if one letter type is overrepresented (either vowels or consonants). We leave α at 0.05. Because we do not focus on a single letter type (either type may be overrepresented), we formulate non-directional hypotheses, first in textual form,

H_1: one letter type is overrepresented,
H_0: no letter type is overrepresented,

next in operationalized form,

H_1: there are more letters of the same type than expected by chance,
H_0: there are no more letters of the same type than expected by chance,

and finally in statistical form:

H_1: $p_V > 0.5$ or $p_C > 0.5$,
H_0: $p_V = p_C = 0.5$.

Suppose that we draw again a four-letter word. The probability of finding four letters of the same type is obtained by adding the probability of finding four consonants (the first row in Tab. 8.5) and the probability of finding four vowels (the last row in Tab. 8.5): $0.0625 + 0.0625 = 0.125$. Again, $p > \alpha$, and H_1 is rejected. Again, the number of trials is too small to make a significant conclusion.

[9] In `qbinom`, `q` stands for "quantile". The quantile function is the inverse of the cumulative distribution function. See Dalgaard (2008, pp. 63–64) for more explanations.

Let us ask the same question as above, this time with a non-directional H_1, namely: how many letters of the same type can you draw until you decide that the type is overrepresented in the text? We use `qbinom()` again, but we leave out the `lower.tail` argument and divide α by two because H_1 is two-tailed.

```
> qbinom(0.05/2, 50, 0.5)
[1] 18
```

Only when we draw 18 or more letters of the same type out of 50 draws can we reject H_0 with a 5% level of significance and conclude that the letter type is overrepresented.

8.8.2 Continuous Distributions

The most famous continuous distribution is the normal distribution. It is the distribution that characterizes most biological phenomena such as people's heights, IQ scores, the gestation period of mammals, the size of potatoes, the duration of lightning, etc. The normal distribution is far from being the norm in corpus linguistics (as opposed to, say, psycholinguistics). I will therefore only outline the key aspects of this continuous distribution.

When a variable is normally distributed, its distribution has the shape of a bell curve (Fig. 8.4). The main reason for this shape is the so-called "central-limit theorem". According to this theorem, random variables formed by adding many random effects will be approximately normally distributed.

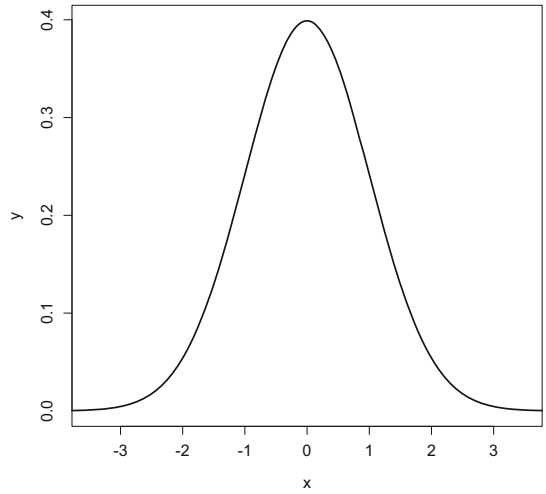


Fig. 8.4: A normal density curve

The normal distribution (and its associated curve) has mathematical properties. It is completely determined by the mean (μ) and the standard deviation (σ). A normal curve is symmetric about and centered at the mean, mode, and median of the variable. Its spread is indexed on σ: the larger σ, the more spread out the distribution, and the flatter the curve. Almost all the observations of the variable are found within 3σ on either side of the mean.

The data set `cars.rds` contains the length measurements (in inches) of 387 cars.[10]

```
> rm(list=ls(all=TRUE))
> data <- readRDS("/CLSR/chap8/cars.rds")
> str(data)
'data.frame': 387 obs. of  1 variable:
 $ length: int  167 153 183 183 183 174 174 168 168 168 ...
```

We want to see if the data are normally distributed. We plot a histogram of the frequency distribution. We obtain Fig. 8.5a.

```
> hist(data$length, xlab="length of cars", main="")
```

Roughly speaking, the bins display a certain symmetry on either side of the highest bin. In Fig. 8.5b, we plot probability densities instead of frequencies. This is done by adding an argument: `prob=TRUE` (or `freq=FALSE`). With `lines()`, we fit a density curve. With `adjust=2`, the approximation of the density curve is "smoother" than if you leave the default setting (`adjust=1`). The result is a neat bell-shaped curve.

```
> hist(data$length, prob=TRUE, xlab="length of cars", main="") # or freq=FALSE instead of prob=TRUE
> lines(density(data$length, adjust=2), col="blue", lwd=2)
```

The area delimited by the normal density curve is used to determine if an observed value falls within the acceptance region ($p < 0.05$). The whole area under the density curve represents 100%. If α is set to 0.05, this area is amputated from 2×0.025 (0.025 at each tail). The acceptance region represents 95% of the total area under the curve.[11] Let us visualize this area. First, we define the starting point and the end point of the area on the horizontal axis with `quantile()`.

```
> q025 <- quantile(data$length, .025)
> q975 <- quantile(data$length, .975)
```

Next, we vectorize the density of the data.

```
> dens <- density(data$length, adjust = 2)
```

This allows us to find the x-axis coordinates with respect to density. With `which()`, we find those points on the curve that are above 0.025 and below 0.975 (we exclude α, i.e. 0.05).

```
> x1 <- min(which(dens$x >  q025))
> x2 <- max(which(dens$x <  q975))
```

Finally, we plot the density curve,

```
> plot(density(data$length, adjust = 2), col="blue", lwd=2)
```

and add (`with()`) shading to color the defined region of acceptance with `polygon()`.

```
> with(dens, polygon(x = c(x[c(x1, x1:x2, x2)]), y = c(0, y[x1:x2], 0), col="grey"))
```

[10] I am using just one part of a data set posted by Roger W. Johnson (Department of Mathematics and Computer Science, South Dakota School of Mines and Technology) on the *Journal of Statistics Education* Data Archive (http://www.amstat.org/publications/jse/jse_data_archive.htm). For more information on the data set, please go to http://www.amstat.org/publications/jse/datasets/04cars.txt.

[11] 95% of the data points are in this area.

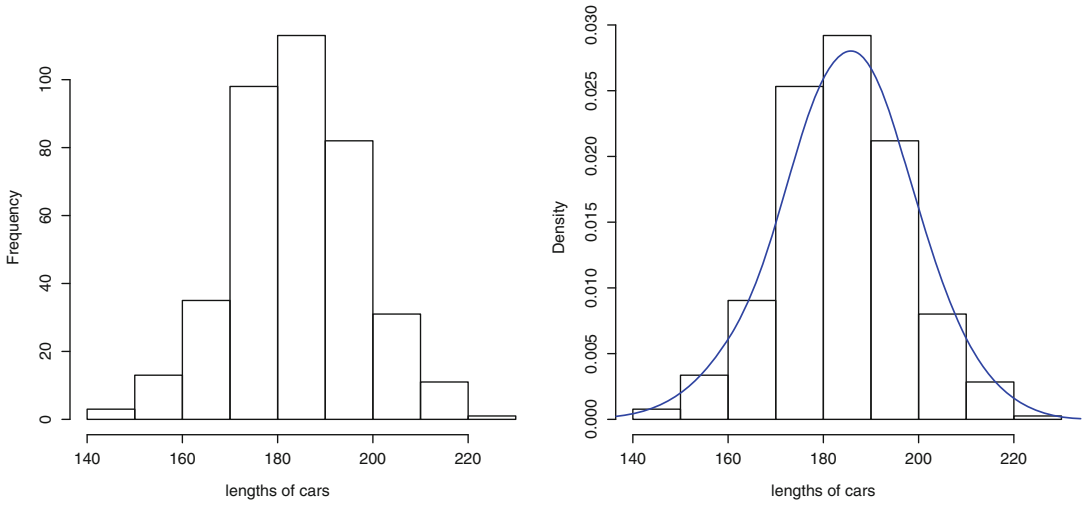


Fig. 8.5: Plotting the normally distributed population of car lengths. (**a**) Frequency (`hist(data$length)`); (**b**) density (`hist(data$length, prob=TRUE)`)

We obtain Fig. 8.6.

However, even when the shape looks normal, you should refrain from assuming normality based on the sole observation of a curve. For a better assessment, you should also make a so-called Q-Q plot (for "quantile versus quantile"). The observed values (y-axis) are plotted against the theoretical values (x-axis). If the data are normally distributed, the data points are distributed along a straight line in a Q-Q plot. Such is the case with our data (Fig. 8.7a).

```
> qqnorm(data$length)
```

Sometimes, even a Q-Q plot will be tricky to interpret. You will see a straight line, but might be mislead to interpret the data as normally distributed because of the presence of outliers. In Fig. 8.7a, believe it or not but the data are not normally distributed because of the extreme data points at both tails.[12]

Ultimately, you should use a test of normality such as the Shapiro-Wilk test. You formulate two hypotheses:

H_0: the data are normally distributed,
H_1: the data are not normally distributed.

In R, this test is carried out by means of the `shapiro.test()` function.

```
> shapiro.test(data$length)

Shapiro-Wilk normality test

data:  data$length
W = 0.99394, p-value = 0.1256
```

[12] The data set consists of 2000 observations of one variable (the weight of euro coins in gram). See http://www.amstat.org/publications/jse/datasets/euroweight.dat.txt.

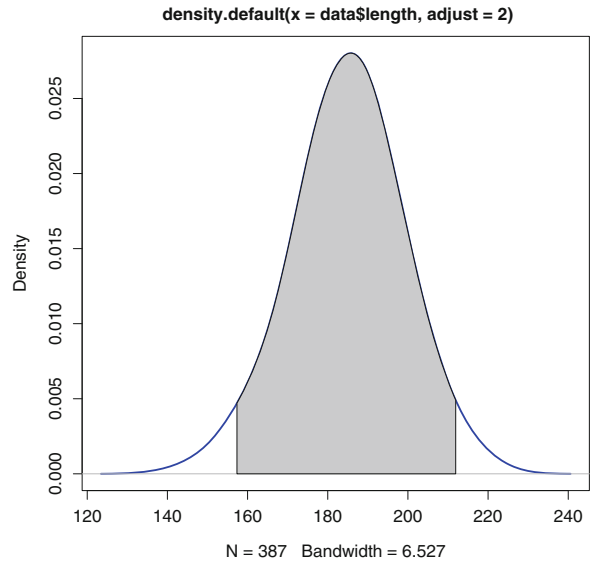


Fig. 8.6: 95% confidence interval for the length (in inches) of cars in `cars.rds`

The test outputs a statistic (W) and a p-value. Here, because $p > 0.05$, you reject H_1 and accept H_0. The data are normally distributed.

In the `cars.rds` example, we have enough data to observe a bell-shaped density curve. This is not always the case. Let us turn to a more linguistic case study. Isel et al. (2003) investigate the auditory

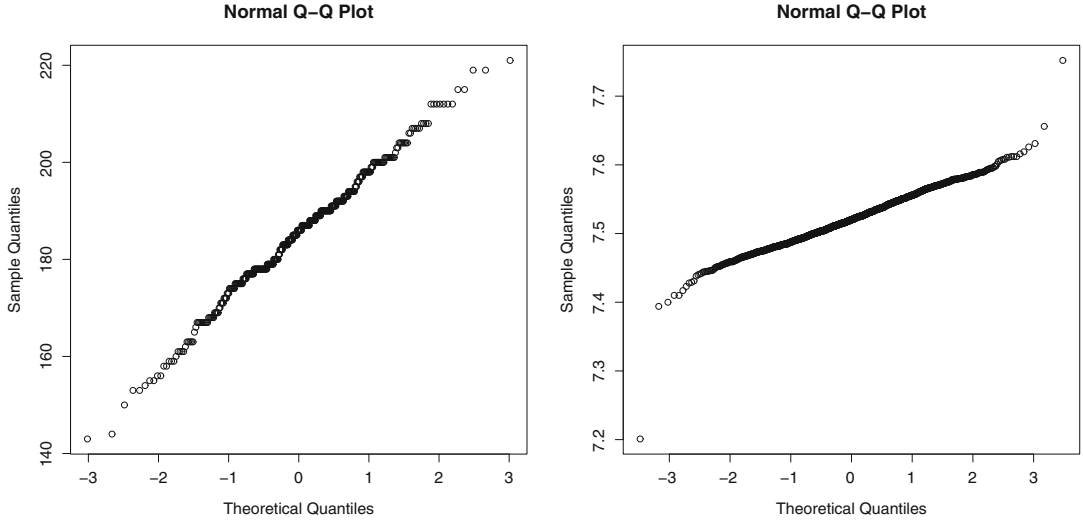


Fig. 8.7: Q-Q plots. (**a**) A Q-Q plot of `data$length`. (**b**) A Q-Q plot of non normally distributed data

processing of German compound words consisting of two nominal constituents. The compound words break down into four categories, depending on the semantic link between the constituents:

- transparent-transparent (T-T): e.g. *Kerkorgel* (church organ),
- opaque-opaque (O-O): e.g. *Klokhuis* (apple core),
- opaque-transparent (O-T), e.g. *Fliegenpilz* (fly mushroom, i.e. toadstool),
- transparent-opaque (T-O): e.g. *Drankorgel* (booze organ, i.e. boozer).

The authors carried out a lexical decision task. Twelve participants were asked to decide if a sequence of letters was a German compound word. The reaction times in milliseconds were recorded. The data set is available from the `chap8` folder: `german.compounds.rds`.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap8/german.compounds.rds") # Windows
> data <- readRDS("/CLSR/chap8/german.compounds.rds") # Mac
```

There are four experimental conditions (one condition per compound category).

```
> str(data)
'data.frame': 12 obs. of  4 variables:
 $ TT: int  519 532 511 617 540 626 549 531 539 517 ...
 $ OO: int  517 536 537 592 558 567 520 524 554 480 ...
 $ OT: int  509 541 541 604 566 623 535 530 535 505 ...
 $ TO: int  520 509 547 625 575 567 553 524 514 493 ...
```

By means of a `for` loop, we plot the histogram and density curve of each variable. We obtain Fig. 8.8.

```
> par(mfrow=c(2,4))
> for (i in 1:ncol(data)) {
+   hist(data[,i], prob=TRUE, xlab="reaction times", main=colnames(data)[i])
+   lines(density(data[,i], adjust=2), col="blue", lwd=2)
+ }
```

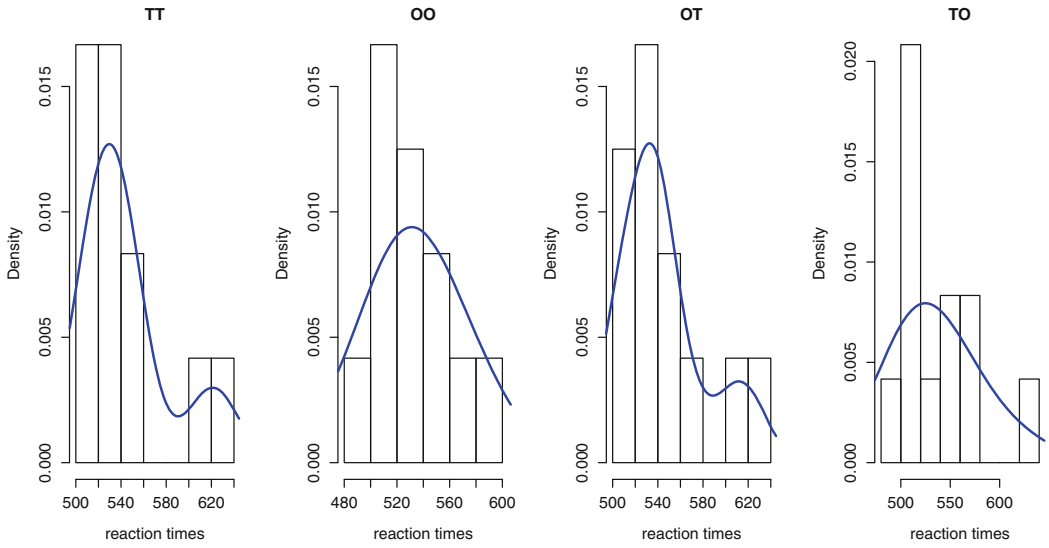


Fig. 8.8: Histograms and density curves for the four variables in `german.compounds.rds`

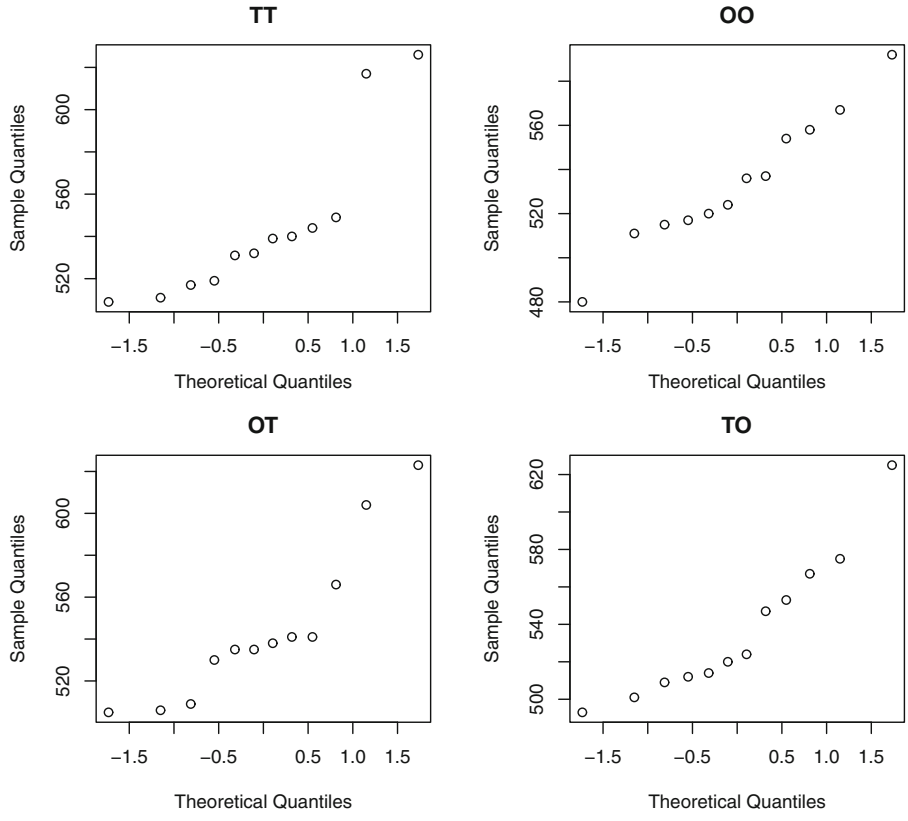


Fig. 8.9: Q-Q plots of the four variables in `german.compounds.rds`

Upon inspection, we see that density curves of `TT` and `OT` deviate from what we expect from a normal distribution (no bell-shaped curve). We suspect that `OO` and `TO` are normally distributed, even though we do not have enough data to see the full bell-shaped curve. Outliers are easy to spot in the Q-Q plots of `TT` and `OT` (Fig. 8.9)

```
> par(mfrow=c(2,2))
> for (i in 1:ncol(data)){
+   qqnorm(data[,i], main=colnames(data)[i])
+ }
```

This is confirmed when we run a Shapiro-Wilk test for each of the four variables by means of a `for` loop.

```
> for (i in 1:ncol(data)) {
+   print(shapiro.test(data[,i]))
+ }

Shapiro-Wilk normality test

data:  data[, i]
W = 0.77683, p-value = 0.005169


Shapiro-Wilk normality test
```

```
data:  data[, i]
W = 0.97351, p-value = 0.9439


Shapiro-Wilk normality test

data:  data[, i]
W = 0.85482, p-value = 0.04212


Shapiro-Wilk normality test

data:  data[, i]
W = 0.89843, p-value = 0.1514
```

For TT and OT, $p < 0.05$ (0.005169 and 0.04212 respectively). We reject H_0 and conclude that these variables are not normally distributed. For OO and OT, $p > 0.05$ (0.9439 and 0.1514 respectively). We accept H_0 and conclude that these variables are normally distributed.

Knowing whether a variable is normally distributed is important because you can conduct some tests that are specifically designed for normal data. One of these tests is the t-test. The t statistic follows a t distribution. It is very similar to the normal distribution (Fig. 8.10). The t distribution is determined by its degrees of freedom (see Sect. 8.9).

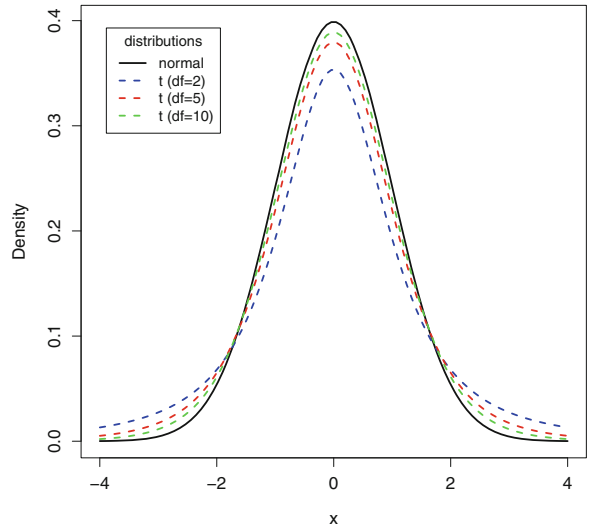


Fig. 8.10: A comparison of the t distribution (with two, five, and ten degrees of freedom) and the normal distribution

Suppose you focus on opaque-opaque compounds. The mean reaction time for this variable is 534.25 ms.

```
> mean(data$OO)
[1] 534.25
```

Suppose also that the study replicates a previous experiment in similar conditions where the linguist found that the mean reaction time was 524.25 ms. One question you might ask is whether the mean you observed is significantly different from the mean that your colleague observed. Given that the two samples are independent you can use a two-tailed one-sample *t*-test to answer this question with `t.test()`. Its first argument is the vector of reaction times for `OO` and the second argument (`mu`) is the mean observed in the previous study.

```
> t.test(data$OO, mu=524.25)

One Sample t-test

data:  data$OO
t = 1.1587, df = 11, p-value = 0.2711
alternative hypothesis: true mean is not equal to 524.25
95 percent confidence interval:
 515.2542 553.2458
sample estimates:
mean of x
   534.25
```

The hypotheses that you are testing are the following:

H_0: the mean you have obtained is equal to 524.25;
H_1: the mean you have obtained is not equal to 524.25.

The test statistic follows a *t* distribution with 11 degrees of freedom. The *p*-value associated with the statistic is above 0.05. It tells you that H_0 should be rejected. The mean that you have observed is not equal to 524.25 (of course, you know that already) but it does not differ significantly from the mean of the previous study. The 95% confidence interval specifies a range of values for which no significant difference with the mean observed in the previous study can be stated.[13]

If we compare the means observed in `data$OO` and `data$OT`, we are comparing two sets of data from the same group, i.e. scores from the same group under two conditions. The data sets are normally distributed, dependent, and paired. In this case, we use a paired *t*-test.[14]

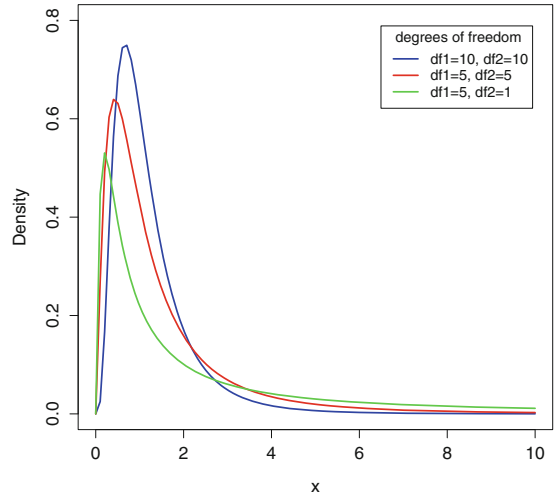
```
> t.test(data$OO, data$OT, paired=TRUE)

Paired t-test

data:  data$OO and data$OT
t = -1.8446, df = 11, p-value = 0.09216
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -22.297323   1.963989
sample estimates:
mean of the differences
              -10.16667
```

The paired *t*-test outputs the difference between the two means. The *p*-value is above 0.05 and the confidence interval includes 0, which reflects an insignificant result. The two means are not significantly different.

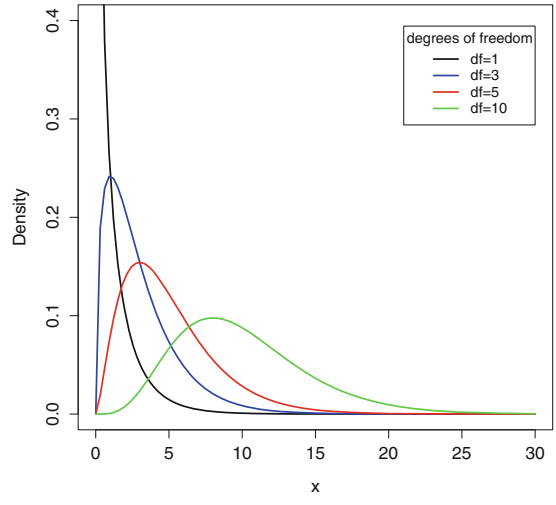
[13] If the two independent variables have differing variances, you should consider running Welch's *t*-test.

[14] If the data are not normally distributed and the samples dependent, run a Wilcoxon-test for independent samples (`wilcox.test()`).

There are other continuous distributions: the F distribution (Fig. 8.11a) and the χ^2 distribution (Fig. 8.11b). The F distribution has two defining parameters, namely two degrees of freedom. The χ^2 distribution takes one parameter: one degree of freedom. The above will become clear when we mention each distribution later on.



(a)



(b)

Fig. 8.11: The F and χ^2 distributions. (**a**) F distributions with three different combinations of degrees of freedom. (**b**) χ^2 distributions with four different degrees of freedom

8.9 The χ^2 Test

Now that you are familiar with hypothesis testing and distributions, it is time to become acquainted with one of the most important tests in statistics and quantitative corpus linguistics: the χ^2 test. It is named after the Greek letter χ (pronounced [kai]). The χ^2 test of independence is used to determine whether there is a significant association between two categorical variables from a single population.

8.9.1 A Case Study: The Quotative System in British and Canadian Youth

Tagliamonte and Hudson (1999) compare the quotative systems of contemporary British and Canadian youth. Quotatives are verbs used to report segments of dialogues subjectively. Here are three examples from the Strathy corpus (*Corpus of Canadian English: 50 million words, 1920s-2000s*).

(5) She**'s like**, I guess I have to. (Strathy—2002-SPOK-CBC-TV)

(6) I said 'Why did you move?', and she **goes** 'Didn't you hear?' (Strathy—2004-ACAD-TheCanGeographer)

(7) 'You've got to let us get a picture with you,' and he **says**, 'Absolutely.' (Strathy—2007-NEWS-CBCnews.ca)

The personal-experience narratives of 44 British speakers and 23 Canadian speakers were recorded in 1995 and 1996. The informants are university students between 18 and 28 years of age. Tab. 8.6 is adapted from Tagliamonte and Hudson (1999, p. 158). It shows the respective overall distribution of quotatives found in the British and Canadian corpora. The British corpus contains 665 quotatives, and the Canadian corpus 612 quotatives. The category "misc" groups various verbs such as *decide*, *tell*, *yell*, *ask*, *scream*, etc.

Table 8.6: The distribution of quotatives in a British corpus and a Canadian corpus (adapted from Tagliamonte and Hudson 1999, 158)

quotatives	British English	Canadian English
say	209	219
go	120	135
be like	120	79
think	123	27
zero	66	123
be (just)	11	5
misc	16	24

In the original article, the table was interpreted on the basis of raw frequencies and percentages. Our goal is twofold. Firstly, we want to check whether the use of a quotative is independent from the variety of English with a type I error set to 5%. Secondly, if the use of a quotative depends on the variety of English, we wish to know which quotatives depend the most on the variety of English.

In your `chap8` subfolder, you will find Tab. 8.6 in R data file format (`quotatives.rds`). After clearing R's memory, import the data set into R, and inspect it.

```
> rm(list=ls(all=T))
> data <- readRDS("/CLSR/chap8/quotatives.rds")
> data
        British_English Canadian_English
say                 209              219
go                  120              135
be_like             120               79
think               123               27
zero                 66              123
be_just              11                5
misc                 16               24
```

Because the table contains frequency counts, with which we are going to perform calculations, let us convert the data frame into a matrix.

```
> data <- as.matrix(data)
```

You may want to visualize the contingency table by means of a barplot. To compare British English and Canadian English more easily, let us make two separate plots on top of each other. With `par()`, we can combine them.

```
> par(mfrow=c(2,1)) # prepare an empty plotting window with 2 rows and 1 column
> barplot(data[,1], main="British English") # plot the first column of the table
> barplot(data[,2], main="Canadian English") # plot the second column of the table
```

You obtain Fig. 8.12. The stacked barplots reveal some similarities with respect to the choice of *say*, *go*, *be just*, and *misc*. They also reveal some differences with respect to the choice of *be like*, *think*, and *zero*. We should refrain from interpreting these plots further because (a) we do not know yet whether the choice of quotatives and the variety of English are interdependent, and (b) these plots are based on raw frequencies, regardless of the marginal totals (i.e. the sum totals of rows and columns) and cannot tell us how surprised or not surprised we should be when assessing each number.

To determine whether the occurrence of quotatives depends on the variety of English, you compute a χ^2 test of independence, whose assumptions are identical to the χ^2 test of goodness-of-fit.

Because we are interested in researching if two categorical variables are interdependent, we formulate the hypotheses as follows:

H_0: the choice of quotatives and the variety of English are independent;
H_1: the choice of quotatives and the variety of English are interdependent.

One calculates the χ^2 value of a cell in the i^{th} row and the j^{th} column as follows:

$$\chi^2_{i,j} = \frac{(E_{i,j} - O_{i,j})^2}{E_{i,j}} \tag{8.14}$$

where $E_{i,j}$ is the expected frequency for cell i, j and $O_{i,j}$ is the observed frequency for cell i, j. The χ^2 statistic of the whole table, χ^2 is the sum of the χ^2 values of all cells.

$$\chi^2 = \sum_{i=1}^{n} \frac{(E - O)^2}{E} \tag{8.15}$$

Thanks to `chisq.test()`, you do not need to calculate χ^2 manually. This function outputs a list. The first element of the list contains the χ^2 value (`statistic`).

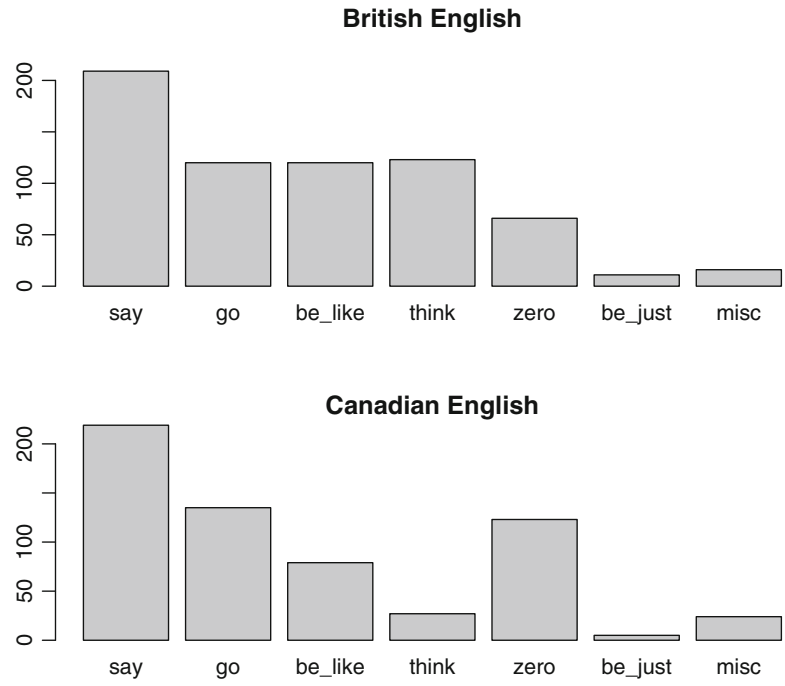


Fig. 8.12: Two barplots showing the distribution of quotatives with respect to the variety of English

```
> chisq.test(data)$statistic
X-squared
 89.99905
```

The test tells us that $\chi^2 = 89.999$, which does not say much in itself. We want to know whether the association is statistically significant. Back in the old days, the statistical significance of the χ^2 value had to be infered from a χ^2 distribution table such as Tab. A.1 (Sect. A.2.3). In this table, you have the critical χ^2 values for six levels of significance: $p = 0.1$, $p = 0.05$, $p = 0.025$, $p = 0.01$, $p = 0.005$, and $p = 0.001$. The smaller the p-value, the higher the level of significance. The critical values for the χ^2 statistic are the values that we can accept for a given significance level. To use this table, you need to know the degree(s) of freedom of your data table (see the leftmost column). The degree(s) of freedom is the number of values that you need to know in order to complete a data table in which only the marginal totals are given. For example, given the marginal totals only, you cannot infer the cell values in Tab. 8.7. You can only infer some values if you are given the marginal totals and one cell value. However, given the marginal totals plus two cell values you can infer the values of all the other cells. For example, if you know that $c = 12$ and $f = 11$, you can derive the values of the other cells very easily:

$$d = 27 - c = 27 - 12 = 15$$
$$e = 14 - f = 14 - 11 = 3$$
$$b = 19 - f = 19 - 11 = 8$$
$$a = 18 - b = 18 - 8 = 10$$

Because you need two cell values to complete the table, the table has two degrees of freedom. Rather than finding df heuristically, which is tedious for large tables, you obtain the number of degrees of freedom with the following formula:

$$\text{df} = (\text{number of rows} - 1) \times (\text{number of columns} - 1). \tag{8.16}$$

Table 8.7: An example table with marginal totals only

	column 1	column 2	total
row 1	a	b	18
row 2	c	d	27
row 3	e	f	14
total	40	19	59

In R, we obtain df thanks to `nrow()` and `ncol()`:

```
> df <- (nrow(data)-1)*(ncol(data)-1)
> df
[1] 6
```

There are six degrees of freedom in Tab. 8.6. With this information, you may now turn to Tab. A.1, partly reproduced in Tab. 8.8. You test your observed χ^2 value for significance by comparing it to the critical χ^2 value observed at the intersection of the number of degrees of freedom in your data table and the desired level of significance. If you set the level of significance to $p = 0.05$ (i.e. 5%), then $\chi^2_{critical} = 12.5916$. Your observed χ^2 value is larger than $\chi^2_{critical}$ with a significance level of $p = 0.05$. In fact, it is also larger than $\chi^2_{critical}$ for $p = 0.001$. This means that a χ^2 value of 89.999 is very unlikely to be observed in a comparable sample if the choice of a quotative were independent from the variety of English. We can safely reject the hypothesis of independence and conclude that the choice of a quotative depends on the variety of English or that the variety of English is characterized by different choices of quotatives.

Table 8.8: Finding the critical value of χ^2

degrees of freedom	$p = 0.1$	$p = 0.05$	$p = 0.025$	$p = 0.01$	$p = 0.005$	$p = 0.001$
...	...	...	...	...	...	...
4	7.7794	9.4877	11.1433	13.2767	14.8603	18.4668
5	9.2364	11.0705	12.8325	15.0863	16.7496	20.515
6	10.6446	**12.5916**	14.4494	16.8119	18.5476	22.4577
7	12.017	14.0671	16.0128	18.4753	20.2777	24.3219
8	13.3616	15.5073	17.5345	20.0902	21.955	26.1245
...	...	...	...	...	...	...

Thanks to the `chisq.test()` function in R, you do not need to use a χ^2 distribution table anymore.

```
> test <- chisq.test(data)
> test

Pearson's Chi-squared test

data:  data
X-squared = 89.999, df = 6, p-value < 2.2e-16
```

The function indicates that the p-value is close to zero. We have no doubt about the significance level of the χ^2 value.

We know that the row and column variables are associated. We also know that this association is statistically significant. Yet, we have no idea about the intensity of this association. Unfortunately, we cannot use the χ^2 value to assess the magnitude of the association because χ^2 depends on the sample size. Indeed, if we multiply each cell by ten, this will affect the value of χ^2 accordingly, but it will not increase the intensity of the association.

```
> chisq.test(data*10)

Pearson's Chi-squared test

data:  data * 10
X-squared = 899.99, df = 6, p-value < 2.2e-16
```

Fortunately, several measures can capture the intensity of the association. The most common measures are: Phi (ϕ), Pearson's contingency coefficient, and Cramér's V. All these measures are derived from χ^2. They ranges from 0 (no association) to 1 (perfect association). They have the desirable property of scale invariance: if the sample size increases, the output of these measures does not change as long as values in the table change the same relative to each other.

ϕ is generally used for 2×2 matrices (i.e. matrices with two rows and two columns). It is computed by taking the square root of the χ^2 statistic divided by the sample size (N):

$$\phi = \sqrt{\frac{\chi^2}{N}} \tag{8.17}$$

The χ^2 value is in the `test` object, under `statistic`. N is the sum of all observations. It is obtained by applying `sum()` to the matrix.

```
> chisq <- as.vector(test$statistic)
> N <- sum(data)
```

With this information at hand, ϕ is very easy to calculate. It is not recommended to calculate the ϕ statistic when the matrix is larger than 2×2 because you may get $\phi > 1$, thus going beyond the desired range between 0 and 1. For this reason, let us not display the result of ϕ here.

```
> phi <- sqrt(chisq/N)
```

Pearson's contingency coefficient C is obtained by taking the square root of the χ^2 statistic divided by the sum of the sum of all observations and χ^2.

$$\text{Pearson's coefficient} = \sqrt{\frac{\chi^2}{N+\chi^2}} \tag{8.18}$$

Again, it is very easy to compute in R.

```
> PCC <- sqrt(chisq/(sum(data)+chisq))
> PCC
[1] 0.2565871
```

C is generally used when the matrix has an equal number of rows and columns, which is not the case here. When a matrix is larger than 2×2 and when the number of row variables differs from the number of column variables, Cramér's V is preferred. V is very close to ϕ. It is computed by taking the square root of the χ^2 statistic divided by the product of the sum of all observations and the number of columns minus one

$$\text{Cramér's } V = \sqrt{\frac{\chi^2}{N(k-1)}}. \tag{8.19}$$

Take the number of possible values of the row variables and the column variables: k is the smaller of the two numbers. Because our matrix is a 7×2 table, $k = 2$. For larger tables, you obtain k with the `min()` function.

```
> k <- min(ncol(data), nrow(data))
> k
[1] 2
```

You may now compute Cramér's V as follows.

```
> V <- sqrt(chisq/(N*(k-1)))
> V
[1] 0.2654749
```

The value of V in this case indicates a not-so-strong yet non-negligible association between the rows and the columns. A high value of V would be surprising given the nature of the phenomenon under investigation. Because V is very simple to calculate, it does not have a base-R function. If you do not want to calculate it yourself, custom-made functions are available from a bunch of packages: e.g. `CramerV()` from the `DescTools` package or `cramersV()` from the `lsr` package. Cramér's V is calculated automatically when you run a χ^2 test of independence with `assocstats()` from the `vcd` package.

Having answered the first question, let us now turn to the second one. The content of the `test` object is revealed thanks to `str(test)`. It contains several kinds of useful information. The expected frequency for each cell is obtained as follows:

```
> test$expected
        British_English Canadian_English
say          222.881754       205.118246
go           132.791699       122.208301
be_like      103.629601        95.370399
think         78.112764        71.887236
zero          98.422083        90.577917
be_just        8.332028         7.667972
misc          20.830070        19.169930
```

This line of code outputs the matrix we would have if there were no association between the choice of quotative and the variety of English but still the same marginal totals. By comparing the matrix of expected frequencies to the original matrix of raw frequencies, we can make an informed judgment as to what quotative occurs more or less often than expected in each variety of English.

Because comparing matrices manually is tedious, a much quicker way to find this out involves the Pearson residuals. If a Pearson residual is positive, then the corresponding observed frequency is greater than its

expected frequency. If a Pearson residual is negative, then the corresponding observed frequency is smaller than its expected frequency. Second, the more the Pearson residual deviates from 0, the stronger the effect. For a cell in the i^{th} row and the j^{th} column, you calculate Pearson residuals as follows:

$$\text{Pearson residuals} = \frac{O_{ij} - E_{ij}}{\sqrt{E_{ij}}} \tag{8.20}$$

Fortunately, with the `chisq.test()` function, you do not need to calculate Pearson residuals for all cells.

```
> test$residuals
        British_English Canadian_English
say          -0.9298376        0.9692643
go           -1.1100506        1.1571186
be_like       1.6081160       -1.6763028
think         5.0788087       -5.2941589
zero         -3.2680947        3.4066675
be_just       0.9242849       -0.9634762
misc         -1.0582983        1.1031719
```

R allows you to visualize the table of Pearson residuals by means of a Cohen-Friendly association plot (Cohen 1980; Friendly 2000). You can generate this plot with `assocplot()` in base R (Fig. 8.13).

```
> assocplot(data)
```

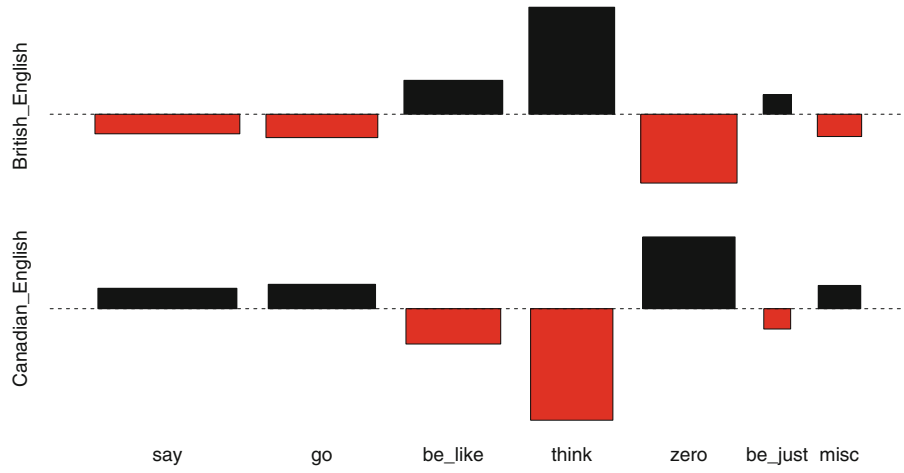


Fig. 8.13: Association plot made with `assocplot()`, from base R

Each cell of `data` is represented by a tile whose height is proportional to the corresponding Pearson residual and whose width is proportional to the square root of the expected counts. The area of the box is therefore proportional to the difference between observed and expected frequencies. The dotted line is the baseline. It represents independence. If the observed frequency of a cell is greater than its expected frequency, the tile appears above the baseline and is shaded black. If the observed frequency of a cell is smaller than its expected frequency, the tile appears below the baseline and is shaded red. The two largest tiles (*think* and *zero*) are easy to discern, but assessing the status of the third biggest tile is trickier.

Judging that the relative size of the tiles is insufficient to assess association intensity, Zeileis et al. (2007) improved `assocplot()` by proposing a selective color shading based on Pearson residuals. Fig. 8.14 is obtained with the function `assoc()`, from the `vcd` package. Note that this plot is obtained by transposing the rows and columns of `data`, so as to facilitate the comparison with the plot in Fig. 8.13.

```
> library(vcd)
> assoc(t(data), shade=TRUE)
```

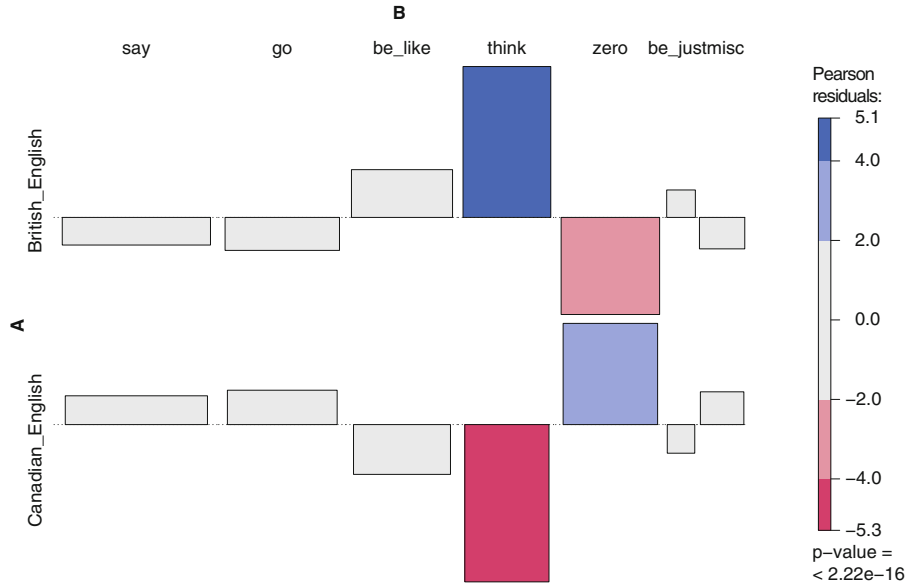


Fig. 8.14: Association plot made with `assoc()`, from the `vcd` package

No tile whose residual is lower than 2 or higher than -2 is shaded. This association plot confirms what the inspection of `test$residuals` shows: the quotatives that contribute the most to the non-independence between the quotatives and the variety of English are *think* and *zero*.

8.10 The Fisher Exact Test of Independence

Unlike the χ^2 test of independence, which calculates the p-value from the asymptotic χ^2 distribution of the test statistic, the Fisher exact test computes exact p-values (as its name indicates). For this reason, this test is computationally expensive for large tables (larger than 2×2), and may even crash if the frequency values are too large.

```
> fisher.test(data)

Error in fisher.test(data):  FEXACT error 7.
LDSTP is too small for this problem.
Try increasing the size of the workspace.
```

In this case, rather than calculate all the exact p-values, you may simulate them by means of what is known as Monte Carlo simulations by setting the `simulate.p.value` argument to `TRUE`.

```
> fisher.test(data, simulate.p.value = TRUE)

Fisher's Exact Test for Count Data with simulated p-value (based on 2000 replicates)

data:  data
p-value = 0.0004998
alternative hypothesis: two.sided
```

By default, the number of replicates is 2000. You may change this number via the `B` argument.

```
> fisher.test(data, simulate.p.value = TRUE, B = 5000)

Fisher's Exact Test for Count Data with simulated p-value (based on 5000 replicates)

data:  data
p-value = 2e-04
alternative hypothesis: two.sided
```

Even if `fisher.test()` allows for simulations, bear in mind that when the sample size is large, a parametric test will work just fine. The `fisher.test()` function outputs a p-value that can be interpreted as the likelihood of obtaining a table with significantly larger differences.

8.11 Correlation

Correlation is the association between two random variables (x and y). It is measured with a correlation coefficient that is comprised between -1 and $+1$. If the correlation coefficient is equal to -1 or 1, the correlation is perfect (Fig. 8.15a, b): knowing the value of x (or y) allows you to predict the exact value of y (or x). In linguistics, very few, if any, variables are perfectly correlated with each other. This means that you cannot predict the exact value of one variable given the other (you can estimate it, however). If the coefficient is equal to 0, there is no correlation (Fig. 8.15g). It is rare to find a correlation coefficient equal to 0. Just because you are very likely to find a correlation between variables almost all the time does not mean that the correlation is significant or meaningful.

The closer the coefficient is to -1 or 1, the stronger the correlation. A positive coefficient means that the more x, the more y or the less x, the less y (Fig. 8.15c, e). A negative coefficient means that the more x, the less y, or the less x, the more y (Fig. 8.15d, f). The closer the coefficient is to 0, the weaker the correlation.

Below, I present three correlation measures: Pearson's r, Kendall's τ, and Spearman's ρ. All three measures are computed thanks to one function: `cor()`. Because the formulas are intimidating, and not particularly helpful to non statisticians, I will not discuss them.

8.11.1 Pearson's r

To see to what extent British English and American English were different, Hofland and Johansson (1982) compared word frequencies in the Brown corpus (American English) and the LOB corpus (British English).[15] Following Leech and Fallon (1992) and Oakes (2009), who replicated the experiment, we compare

[15] Chapter 8 of Hofland and Johansson's book contains a frequency list. The list is restricted to the most frequent words.

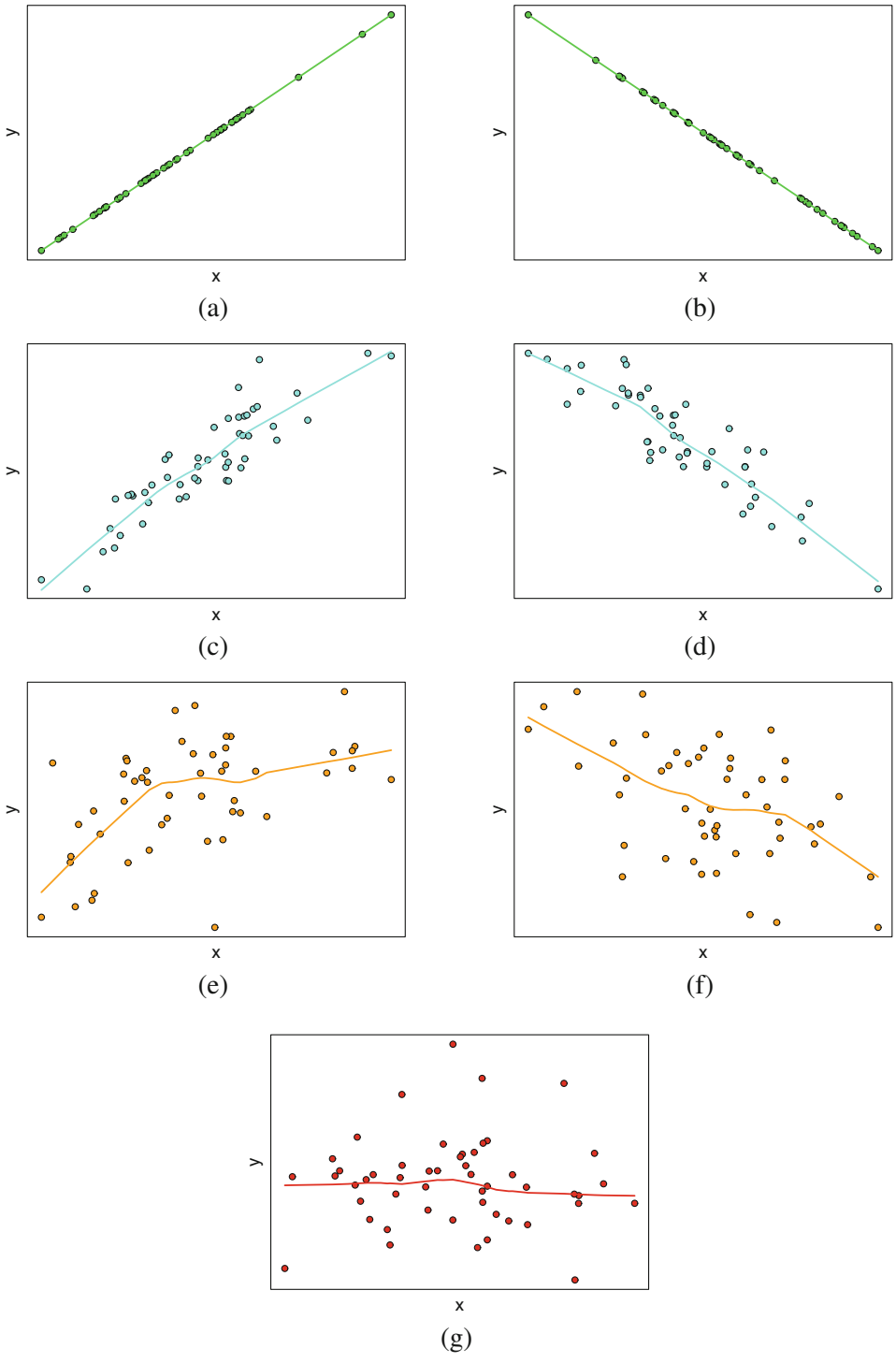


Fig. 8.15: Plotting correlated variables. (**a**) A perfect positive correlation (= 1); (**b**) a perfect negative correlation (= −1); (**c**) a strong positive correlation (= 0.9); (**d**) a strong negative correlation (= −0.9); (**e**) a mild positive correlation (= 0.5); (**f**) a mild negative correlation (= −0.5); (**g**) no correlation (= 0)

the word frequency distributions of the eight text types represented in the BNC (XML Edition). The hypotheses underlying the data set are the following:

H_0: the word distribution in a text type does not correlate with the word distributions in other text types;
H_1: the word distribution in a text type correlates with the word distributions in other text types.

These hypotheses are perhaps too general, but we will specify them afterwards.

The data set `wordfreq_bnc_genres.rds` is available from your `chap5` subfolder. It consists of 33 rows, one for each of the 33 most common words and 8 columns: `ACPROSE` (academic writing), `FICTION` (published fiction), `NEWS` (news and journalism), `NONAC` (published non-fiction), `OTHERPUB` (other published writing), `UNPUB` (unpublished writing), `CONVRSN` (conversation), and `OTHERSP` (other spoken).

```
> rm(list=ls(all=TRUE))
> data <- readRDS("/CLSR/chap5/wordfreq_bnc_genres.rds")
> str(data)
'data.frame': 33 obs. of  8 variables:
 $ ACPROSE : int  2559 1860 1342 1036 918 909 725 415 411 363 ...
 $ CONVRSN : int  2004 1038 893 760 1203 1231 424 348 244 101 ...
 $ FICTION : int  2349 1183 563 657 1170 1160 646 265 322 101 ...
 $ NEWS    : int  254 129 116 84 80 104 65 23 25 36 ...
 $ NONAC   : int  315 198 134 132 208 201 69 49 40 42 ...
 $ OTHERPUB: int  156 95 59 94 115 109 43 24 19 21 ...
 $ OTHERSP : int  267 189 139 125 185 306 14 35 31 27 ...
 $ UNPUB   : int  1418 734 480 522 676 763 314 181 127 80 ...
```

With `plot()`, we can visualize the pairwise correlations between variables (Fig. 8.16).

```
> plot(data)
> plot(data, pch=21, bg="cyan", cex=0.6) # a better-looking output
```

Fig. 8.16 is a matrix of scatterplots. Each scatterplot corresponds to the intersection of two variables. The labels are given in the diagonal of the plot. All correlations are positive. Some are quite strong (`NONAC` and `OTHERPUB`, `NONAC` and `UNPUB`, `UNPUB` and `OTHERPUB`). The detail of these correlations is found is a correlation matrix, which you obtain by running `cor()` on the whole data set.

```
> cor(data)
            ACPROSE   CONVRSN   FICTION      NEWS     NONAC  OTHERPUB   OTHERSP     UNPUB
ACPROSE   1.0000000 0.8373300 0.8882179 0.9655391 0.9165912 0.8802978 0.8086054 0.9345967
CONVRSN   0.8373300 1.0000000 0.9299834 0.8915428 0.8669503 0.8456329 0.9006008 0.8929438
FICTION   0.8882179 0.9299834 1.0000000 0.9135481 0.9076236 0.8848384 0.8626348 0.9453973
NEWS      0.9655391 0.8915428 0.9135481 1.0000000 0.9243687 0.8899582 0.8365828 0.9490039
NONAC     0.9165912 0.8669503 0.9076236 0.9243687 1.0000000 0.9775495 0.9042527 0.9763550
OTHERPUB  0.8802978 0.8456329 0.8848384 0.8899582 0.9775495 1.0000000 0.8855048 0.9482751
OTHERSP   0.8086054 0.9006008 0.8626348 0.8365828 0.9042527 0.8855048 1.0000000 0.8975791
UNPUB     0.9345967 0.8929438 0.9453973 0.9490039 0.9763550 0.9482751 0.8975791 1.0000000
```

By default, the correlation matrix is calculated using Pearson's r (also known as "the Pearson product-moment correlation"), which is rooted in the normal distribution. Therefore, it comes with a major restriction: the data must be normally distributed. Because we are dealing with natural-language word distributions, we are likely to find that the data are not normally distributed. As expected, if you plot each column variable of the data, you will see that the data follows a Zipfian distribution (see Fig. 8.17). Pearson's r is therefore not appropriate in this case.

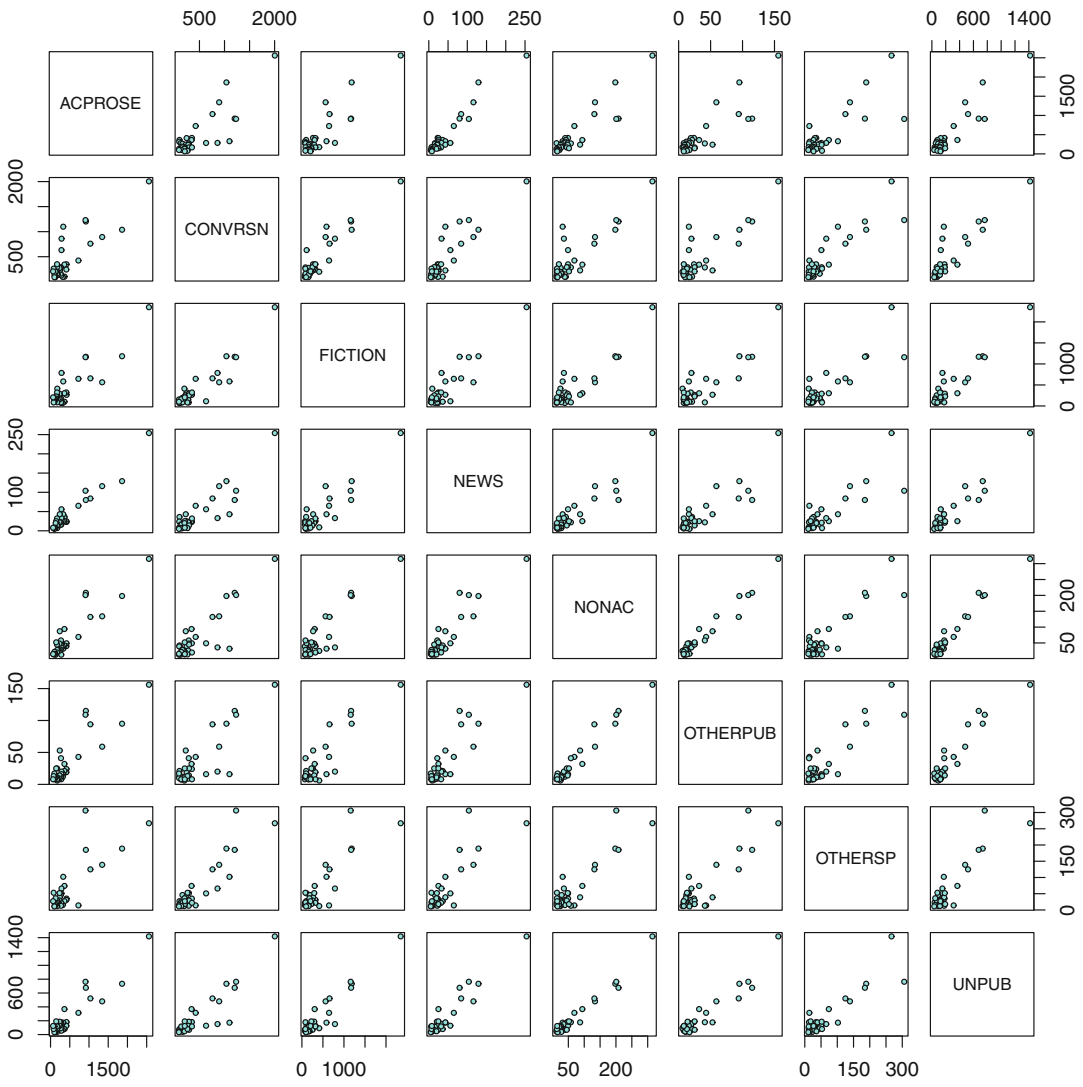


Fig. 8.16: Pairwise correlations between text types in the BNC

8.11.2 Kendall's τ

Fortunately, Kendall's τ ("tau") offers a non-parametric alternative. The τ coefficient is obtained by counting the numbers of concordant and discordant pairs. A pair of points is concordant if the sign of the difference in the x coordinate is the same as the sign of the difference in the y coordinate. It is discordant if the sign of the difference in the x coordinate and the sign of the difference in the y coordinate are different. This method is computationally expensive and should be avoided if your data set contains more than 5000 observations (Dalgaard 2008, section 6.4.3).

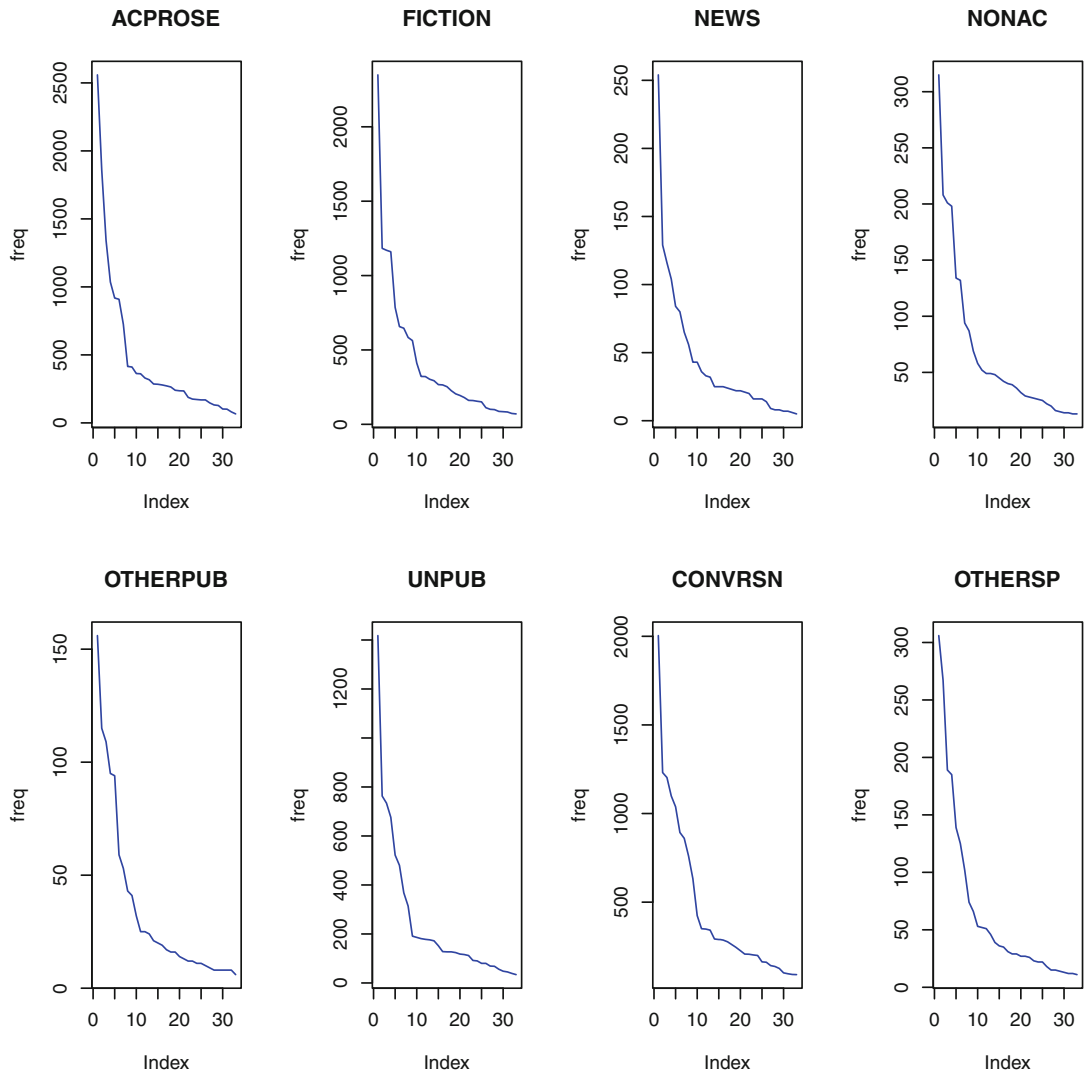


Fig. 8.17: Word distributions for the eight text types in the BNC (33 most frequent common words)

Its main difference is that it takes ranks as input instead of the observations themselves. The input data set is therefore slightly different, as you can see for yourself by loading the data set `wordfreq_bnc_genres_ranks.rds` into R.

```
> data_ranks <- readRDS("/CLSR/chap5/wordfreq_bnc_genres_ranks.rds") # Windows
> data_ranks <- readRDS("/CLSR/chap5/wordfreq_bnc_genres_ranks.rds") # Mac
> str(data_ranks)
```

```
'data.frame': 33 obs. of  8 variables:
 $ ACPROSE : int  6 1 2 5 3 4 12 11 14 32 ...
 $ CONVRSN : int  2 1 5 3 6 8 4 13 7 18 ...
```

```
$ FICTION : int  4 1 2 3 9 6 8 13 5 31 ...
$ NEWS    : int  4 1 2 6 3 5 9 15 12 31 ...
$ NONAC   : int  3 1 4 2 5 6 20 7 19 31 ...
$ OTHERPUB: int  3 1 4 2 6 5 19 10 15 21 ...
$ OTHERSP : int  1 2 3 4 5 6 7 8 9 10 ...
$ UNPUB   : int  1 2 3 4 5 6 7 8 9 10 ...
```

You obtain the correlation matrix based on Kendall's τ by specifying `method="kendall"`.[16]

```
> cor(data_ranks, method="kendall")
        ACPROSE CONVRSN FICTION NEWS NONAC OTHERPUB OTHERSP UNPUB
and           6       2       4    4     3        3       1     1
the           1       1       1    1     1        1       2     2
of            2       5       2    2     4        4       3     3
to            5       3       3    6     2        2       4     4
a             3       6       9    3     5        6       5     5
in            4       8       6    5     6        5       6     6
that         12       4       8    9    20       19       7     7
for          11      13      13   15     7       10       8     8
it           14       7       5   12    19       15       9     9
they         32      18      31   31    31       21      10    10
on           20      15      14   21    14       20      11    11
s            15       9      26    8    12       18      12    12
or           18      26      20   23    21       23      13    13
with         19      20      15   10     8        7      14    14
one          28      23      21   22    30       25      15    15
be            8      12      16   18    13       13      16    16
as            9      19      11   14    17       16      17    17
have         25      14      17   26    15       12      18    18
this         21      17      23   19    24       27      19    19
by           10      30      27   11    16       14      20    20
all          33      21      19   28    29       31      21    21
from         22      27      25   24    23       28      22    22
can          24      25      29   30    22       24      23    23
but          26      11      12   13    27       30      24    24
which        16      29      32   25    32       32      25    25
at           27      22      18   16    11       11      26    26
an           13      31      33   17    18       22      27    27
more         29      32      24   29    33       26      28    28
is            7      10       7    7     9        8      29    29
are          17      16      30   20    10        9      30    30
will         30      28      22   32    26       29      31    31
not          23      24      10   27    25       33      32    32
been         31      33      28   33    28       17      33    33
```

In general, the correlation coefficients based on Kendall's τ are smaller than those based on Pearson's r. Remember that a correlation matrix does not tell you whether the correlation coefficients are statistically significant. The statistical significance of correlation coefficients makes sense in the context of hypothesis testing. Suppose you want to see to what extent the spoken variables are correlated. You formulate two hypotheses:

H_0: the spoken variables are not correlated;
H_1: the spoken variables are correlated.

Once you have operationalized your research question, the hypotheses are converted into the following:

H_0: the word distributions of `CONVRSN` and `OTHERSP` are not correlated;
H_1: the word distributions of `CONVRSN` and `OTHERSP` are correlated.

To obtain the correlation coefficient and its statistical significance, use `cor.test()` and specify `method = "kendall"`. The setting of the `alternative` argument of the function depends on whether your

[16] If your data are normally distributed, leave out this argument which, by default, selects Pearson's r.

hypotheses are directional. If `alternative="two.sided"` (default), your hypotheses do not postulate a specific direction. If `alternative="greater"`, you postulate a positive association in H_1. If `alternative="less"`, you postulate a negative association in H_1. Because we have no reason to assume that the association is positive or negative, H_1 is non-directional. We can therefore leave the default argument setting of `alternative`.

```
> cor.test(data_ranks$CONVRSN, data_ranks$OTHERSP, method="kendall")

          Kendall's rank correlation tau

data:  data_ranks$CONVRSN and data_ranks$OTHERSP
T = 422, p-value = 1.678e-07
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau
0.5984848
```

It turns out that the correlation is not very high but highly significant. In other words, there is a mild but significant positive correlation between the word distribution of `CONVRSN` and `OTHERSP` ($\tau = 0.6$, $T = 422$, and $p = 1.678\text{e-}07$).

8.11.3 Spearman's ρ

The third correlation coefficient is Spearman's ρ ("rho"). It is non parametric and therefore very close to Kendall's τ.

The correlation matrix is obtained by specifying Spearman's ρ in the `method` argument.

```
> cor(data_ranks, method="spearman")
           ACPROSE   CONVRSN   FICTION      NEWS     NONAC  OTHERPUB   OTHERSP     UNPUB
ACPROSE  1.0000000 0.6470588 0.5989305 0.8589572 0.7984626 0.7149064 0.6096257 0.6096257
CONVRSN  0.6470588 1.0000000 0.7854278 0.7703877 0.6985294 0.6527406 0.7593583 0.7593583
FICTION  0.5989305 0.7854278 1.0000000 0.6981952 0.6199866 0.5661765 0.6096257 0.6096257
NEWS     0.8589572 0.7703877 0.6981952 1.0000000 0.8121658 0.7389706 0.6664439 0.6664439
NONAC    0.7984626 0.6985294 0.6199866 0.8121658 1.0000000 0.9070856 0.5875668 0.5875668
OTHERPUB 0.7149064 0.6527406 0.5661765 0.7389706 0.9070856 1.0000000 0.5909091 0.5909091
OTHERSP  0.6096257 0.7593583 0.6096257 0.6664439 0.5875668 0.5909091 1.0000000 1.0000000
UNPUB    0.6096257 0.7593583 0.6096257 0.6664439 0.5875668 0.5909091 1.0000000 1.0000000
```

To obtain correlation coefficients between specific variables, such as `CONVRSN` and `OTHERSP`, we proceed as follows:

```
> cor.test(data_ranks$CONVRSN, data_ranks$OTHERSP, method="spearman")

Spearman's rank correlation rho

data:  data_ranks$CONVRSN and data_ranks$OTHERSP
S = 1440, p-value = 1.325e-06
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.7593583
```

Note that the magnitude of the correlation is different from what we obtain with Kendall's τ.

8.11.4 Correlation Is Not Causation

Mind you, just because two variables are correlated does not imply that one causes the other. Let me take a non-linguistic example. Like me, you may have come across countless articles reporting a correlation between coffee drinking and lung cancer. Although I am no expert in medicine, I find it hard to believe that coffee may impact your lungs. Yet, most scientific articles confirm the correlation.[17] Despite what the correlation suggests, scientists are wary of not jumping to the conclusion that coffee drinking causes lung cancer. Other confounding factors are certainly at play, especially the fact that coffee drinking often goes together with smoking.

As linguists, observing strong and statistically significant correlations between variables should not blind us to the fact that other factors which we may not have taken into account may be at work. Not many predictive techniques manage to bridge the gap between correlation and causation. You should therefore interpret your results with caution.

If you are not convinced by the above warning, I suggest that you visit `http://www.tylervigen.com/spurious-correlations`, a website dedicated to spurious correlations. You will learn that the number of people who drowned by falling into a pool correlates positively with the number of films that American actor Nicolas Cage appeared in ($r = 0.67$).

Exercises

8.1. Simple probabilities
Calculate the probability of finding a consonant and the probability of finding a vowel in Walt Whitman's poem "When I heard the learn'd astronomer" (`/CLSR/chap4/whitman.txt`). Treat the letter "y" as a consonant.

8.2. Simple, joint, and conditional probabilities
Tab. 8.9 shows the distribution of *quite* and *rather* based on whether the adverbs occur in a preadjectival/predeterminer pattern. The table is available at `/CLSR/chap8/ct.rds`.

Table 8.9: The syntactic distribution of *quite* and *rather* in the BNC (XML)

	preadjectival	predeterminer
quite	453	1749
rather	1423	370

(a) Calculate the following probabilities: $P(quite)$, $P(rather)$, $P(preadjectival)$, and $P(predeterminer)$.
(b) Calculate the probability of drawing *rather* in predeterminer position.

[17] See the interesting meta-analysis proposed by Tang et al. (2009). The authors searched PubMed and EMBASE databases between 1966 and January 2009, extracted the data, and found that "combined results indicated a significant positive association between highest coffee intake and lung cancer [relative risk (RR) = 1.27, 95% confidence interval (CI) = 1.04–1.54]".

(c) Calculate the probability of drawing *either* an adverb in predeterminer position *or* the adverb *rather*.
(d) Compare the probability of predeterminer position when it is conditional on the use of *rather* and the probability of *rather* when it is conditional on predeterminer position.
(e) Calculate the following conditional probabilities and explain them briefly: $P(preadjectival|quite)$ and $P(predeterminer|quite)$.

8.3. The distribution of words per line in a tragedy
The case study for this exercise is borrowed from (Muller 1973). *Rodogune* is a seventeenth-century tragedy in five acts by French dramatist Pierre Corneille. At the time, tragedies were highly codified. Use a histogram to determine whether the distribution of words per line is normal. Run a Shapiro-Wilk normality test to confirm your first impression. The text is available here: `/CLSR/chap8/rodogune.txt`.

8.4. Statistical significance
Examine Tab. 8.10. Compare the frequency of word 1 in context 1 to the frequency of word 2 in the same context. Decide if this difference is statistically significant.

Table 8.10: A fictitious contingency table

	context 1	context 2	row totals
word 1	353 (74%)	124 (26%)	477
word 2	157 (78.89%)	42 (21.11%)	199
column totals	510	166	676

8.5. Correlation
The file `stance.rds` is a contingency table that summarizes the distribution of adjective categories across eight text genres in the BNC (XML) (`/CLSR/chap8/stance.rds`). Modify the contigency table with `rank()` and compute the entire matrix of correlations between all variables using Kendall's τ. Determine whether the global correlation is significantly different from zero for the following pairs of variables:

- `ACPROSE` vs. `NEWS`
- `ACPROSE` vs. `NONAC`
- `NEWS` vs. `CONVRSN`

References

Cohen, Ayala. 1980. On the Graphical Display of the Significant Components in a Two-Way Contingency Table. *Communications in Statistics - Theory and Methods* 9 (10), 1025–1041.
Dalgaard, Peter. 2008. *Introductory Statistics with R*, 2nd ed. New York: Springer.
Davies, Mark. *Corpus of Canadian English: 50 Million Words, 1920s–2000s*. http://corpusbyu.edu/can/.

Desagulier, Guillaume. 2015. Forms and Meanings of Intensification: A Multifactorial Comparison of *quite* and *rather*. *Anglophonia* 20. doi:10.4000/anglophonia.558. http://anglophonia.revues.org/558.

Fisher, Ronald Aylmer Sir. 1935. *The Design of Experiments*. Edinburgh, London: Oliver Boyd.

Friendly, Michael. 2000. *Visualizing Categorical Data*. Cary, NC: SAS Publications.

Hofland, Knut, and Stig Johansson. 1982. *Word Frequencies in British and American English*. Bergen: Norwegian Computing Centre for the Humanities.

Isel, Frédéric, Thomas C. Gunter, and Angela D. Friederici. 2003. Prosody-Assisted Head-Driven Access to Spoken German Compounds. *Journal of Experimental Psychology* 29 (2), 277–288. doi:10.1037/0278-7393.29.2.277.

Kilgarriff, Adam. 2005. Language is Never Ever, Ever, Random. *Corpus Linguistics and Linguistic Theory* 1 (2): 263–276. http://dx.doi.org/10.1515/cllt.2005.1.2.263.

Leech, Goeffrey, and Roger Fallon. 1992. Computer corpora – What do they tell us about culture? In *ICAME*, Vol. 16, 29–50.

Muller, Charles. 1973. *Initiation aux méthodes de la statistique linguistique*. Paris: Champion.

Oakes, Michael P. 2009. Corpus Linguistics and Language Variation. In *Contemporary Corpus Linguistics*, ed. Paul Baker, 161–185. London: Bloomsbury, Continuum.

Tagliamonte, Sali, and Rachel Hudson. 1999. *Be like* et al. Beyond America: The Quotative System in British and Canadian Youth. *Journal of Sociolinguistics* 3 (2): 147–172. doi:10.1111/1467-9481.00070.

Tang, Naping, et al. 2009. Coffee Consumption and Risk of Lung Cancer: A Meta-Analysis. *Lung Cancer* 67 (1): 17–22. doi:10.1016/j.lungcan.2009.03.012. http://dx.doi.org/10.1016/j.lungcan.2009.03.012.

Zeileis, Achim, David Meyer, and Kurt Hornik. 2007. Residual-Based Shadings for Visualizing (Conditional) Independence. Journal of Computational and Graphical Statistics 16 (3), 507–525. http://statmath.wu.ac.at/~zeileis/papers/Zeileis+Meyer+Hornik-2007.pdf.

Chapter 9
Association and Productivity

Abstract This chapter covers association measures and productivity measures with respect to lexico-grammatical patterns.

9.1 Introduction

In Sect. 1.2.3.1, we have seen that, because of its theoretical status, frequency has a place of choice in most usage-based theories of language. Before we proceed to measuring association and productivity, let us pause a little and reflect on what these measures are based on: frequency. This concept should not be taken for granted for reasons that I explain below.

Remember that the goal of the mentalist approach to language (Chomsky 1957, 1962) is to uncover the abstract rules that allow speakers to generate an infinity of sentences. This is known as a "top-down" approach, the "top" being the rules, and the "bottom" the generated sentences. In contrast, usage-based approaches to language proceed from the bottom to the top, i.e. inferentially (Bybee 2006, 2010).

Central to the opposition between "top-down" approaches and "bottom-up" approaches is what determines whether an expression is grammatical. Usage-based theories posit that grammaticality is a matter of degree. They have also argued that acceptability judgments with respect to a given expression should be based on the frequency of that expression.[1]

Although the focus on frequency is certainly enticing to corpus linguists, the initial definition of this concept given by first-generation usage-based linguistics is problematic in two respects: it is ambiguous and it is too readily paired with entrenchment, that is to say the moment when a sequence of phonemes, morphemes, or words acquires the status of linguistic units. As Langacker (1987, p. 59) puts it:

> [e]very use of a structure has a positive impact in its degree of entrenchment [...]. With repeated use, a novel structure becomes progressively entrenched, to the point of becoming a unit; moreover, units are variably entrenched depending on the frequency of their occurrence (*driven*, for example, is more entrenched than *thriven*).

According to first-generation usage-based linguistics, frequency is not something tangible that you can empirically apprehend and measure, but something perceived by speakers. Originally, deciding what is entrenched and what is not does not require any form of quantification:

[1] I am using the present perfect deliberately because this view has been toned down recently, as will appear below.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_9

> [i]s there some particular level of entrenchment, with special behavioral significance, that can serve as a nonarbitrary cutoff point in defining units? There are no obvious linguistic grounds for believing so. (Langacker 1987, p. 59)

It is not until recently, with the rise of quantitative corpus linguistics, that first generation usage-based linguists have realized that frequency could be measured:

> [. . .] in principle the degree of entrenchment can be determined empirically. Observed frequency provides one basis for estimating it. (Langacker 2008, p. 238)

Under the impulsion of modern quantitative corpus linguistics, second-generation usage-based linguistics has operationalized frequency and updated its theoretical status. This paradigmatic change is now visible and ready to be applied to areas of linguistics, such as morphological semantics:

> With large samples and appropriate statistical techniques, for example, speaker judgments could help determine whether *ring* 'circular piece of jewelry' and *ring* 'arena' represent alternate senses of a polysemous lexical item [. . .], or whether *computer* is in fact more analyzable than *propeller*. (Langacker 2008, p. 86).

The second issue pertains to the link between high frequency and cognitive entrenchment. This link is not systematic, as shown by Schmid (2010), who claims that salience is also a factor of entrenchment. Orson Welles's *Citizen Kane* provides a good illustration of how salience works. The film opens on the main character, Charles Foster Kane, at the end of his life. Kane dies calling for "Rosebud". It is the job of a reporter to discover who Rosebud is. It appears that Rosebud is (spoiler alert) the trademark of eight-year-old Kane's sled. Although the word is never uttered by Kane in his lifetime as a newspaper magnate, it is heavily entrenched in his mind. Linguistically speaking, the token frequency of a word does not need to be high for the word to be entrenched.

To sum up, frequency should be handled with care. In the paragraphs below, we shall discuss the concept and see how each interpretation should be operationalized.

9.2 Cooccurrence Phenomena

Cooccurrence is the simultaneous occurrence of two linguistic phenomena. Whether the phenomena are just contiguous or meaningfully associated is for association measures to show. Co-occurrence breaks down into collocation, colligation, and, more recently, collostruction.

9.2.1 Collocation

In Chap. 1, I mentioned the Firthian principle according to which "you shall know a word by the company it keeps" (Firth 1957). This is the theoretical foundation of collocation analysis: the meaning of a target word is based on the meaning of the words that surround it. The target word and its context are not merely juxtaposed. Collocates are linked by what Firth calls "mutual expectations" (Firth 1957, p. 181).

In the broad sense, collocation analysis consists in examining the left and/or right contexts of a node word. This is the kind of analysis that motivates a concordance (Sect. 5.2). This sort of collocation analysis works well when it comes to teasing apart homonymous lexemes such as the textbook case: *bank* as a financial institution vs. *bank* as the edge of a river. If *bank* denotes a financial institution, you are very likely to find such collocates as *money*, *crisis*, or *credit*. If *bank* denotes the edge of a river, you are likely to find *river*,

mud, or *fishing* nearby. Of course, collocates do not tell you everything. If a money bank is located by a river, you are in for trouble if you stick to collocation analysis (but this is an extreme case).

In a restricted sense, collocation analysis examines words in specific positions such as the NP position determined by *each* and *every*. Let us see if we can spot semantic differences between *each* and *every* based on their nominal collocates. Let us load the data frame that we compiled in Sect. 5.3.

```
> rm(list=ls(all=TRUE))
> data <- read.delim("C:/CLSR/chap5/df_each_every_bnc_baby.txt", header=TRUE) # Windows
> data <- read.delim("/CLSR/chap5/df_each_every_bnc_baby.txt", header=TRUE) # Mac
```

We extract only the columns that we need with subsetting.

```
> str(data)
'data.frame': 2339 obs. of  8 variables:
 $ corpus.file: Factor w/ 159 levels "A1E.xml","A1F.xml",..: 1 1 1 1 1 1 1 1 1 2 ...
 $ info       : Factor w/ 22 levels "S conv","W ac:humanities arts",..: 10 10 10 10 10 10 10 10
 10 11 ...
 $ mode       : Factor w/ 2 levels "stext","wtext": 2 2 2 2 2 2 2 2 2 2 ...
 $ type       : Factor w/ 5 levels "ACPROSE","CONVRSN",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ exact.match: Factor w/ 912 levels "each action ",..: 252 267 267 97 449 785 136 207 819 592 ...
 $ determiner : Factor w/ 2 levels "each","every": 1 1 1 1 1 2 1 1 2 2 ...
 $ NP         : Factor w/ 597 levels "act","action",..: 334 356 356 127 554 407 173 275 463 147 ...
 $ NP_tag     : Factor w/ 6 levels "NN0","NN1","NN1-AJ0",..: 2 2 2 2 2 2 2 2 2 2 ...
> each.every.np <- data[, c(6,7)] # determiner + NP
> str(each.every.np)
'data.frame': 2339 obs. of  2 variables:
 $ determiner: Factor w/ 2 levels "each","every": 1 1 1 1 1 2 1 1 2 2 ...
 $ NP        : Factor w/ 597 levels "act","action",..: 334 356 356 127 554 407 173 275 463 147 ...
```

We convert the nominal data in `each.every.np$determiner` and `each.every.np$NP` into count data with the `count()` function from the `plyr` package.

```
> library(plyr)
> freqlist <- count(each.every.np, c("NP", "determiner"))
> str(freqlist)
'data.frame': 706 obs. of  3 variables:
 $ NP        : Factor w/ 597 levels "act","action",..: 1 2 2 3 4 5 6 6 7 7 ...
 $ determiner: Factor w/ 2 levels "each","every": 2 1 2 2 1 1 1 2 1 2 ...
 $ freq      : int  2 4 1 1 1 1 2 1 2 3 ...
```

We reorder the resulting frequency list (whose class is data frame) by its columns with `arrange()`. The frequency column is further sorted in descending order with `desc()`.

```
> freqlist <- arrange(freqlist, determiner, desc(freq))
```

To obtain the top collocates of *each*, we subset `freqlist` with a condition: the level of `freqlist$determiner` must be `each`.

```
> head(freqlist[which(freqlist$determiner=="each"),])
     NP determiner freq
1 other       each  419
2  year       each   48
3  case       each   30
4  time       each   28
5   day       each   25
6  side       each   19
```

Because *each* is alphabetically anterior to *every*, the top of `freqlist` contains the most frequent nouns determined by *each*, so conditional subsetting is unnecessary.

```
> head(freqlist)
     NP determiner freq
1 other       each  419
2  year       each   48
3  case       each   30
4  time       each   28
5   day       each   25
6  side       each   19
```

To obtain the top collocates of *every*, we subset `freqlist` with the relevant condition.

```
> head(freqlist[which(freqlist$determiner=="every"),])
       NP determiner freq
376   day      every  139
377  time      every  134
378  year      every   51
379  week      every   34
380 night      every   26
381   man      every   15
```

Each and *every* have several top collocates in common (*day*, *time*, *year*). Both determine nouns that denote time periods. If we want to tease apart *each* and *every*, we are going to have to focus on what nouns are truly specific to each of them. Temporal nouns aside, *each* specifically determines *other*, *side*, and *case* and *every* determines *man*. It seems that *each* focuses on the parts of a whole (other as opposed to ego, one side out of two sides, one case out of several) whereas *every* focuses on the whole (man as a group). Having said that, we should refrain from interpreting the results further because of the low number of collocates that we have taken into account, and because of the choice of raw frequencies used to quantify collocation (see Sect. 9.3.1 below).

9.2.2 Colligation

Colligation is like collocation except that it has a grammatical component. It is therefore the statistically significant co-occurrence of a surface phenomenon (a linguistic form) with a deeper, grammatical phenomenon.

Well known to linguists is the dative alternation, which consists of the prepositional dative (henceforth PD) and the ditransitive constructions (or double-object construction, henceforth DO), as exemplified in (8) and (9) respectively:

(8) John gave the book to Mary. (PD)
S_{AGENT} V O_{THEME} $O_{RECIPIENT}$

(9) John gave Mary the book. (DO)
S_{AGENT} V $O_{RECIPIENT}$ O_{THEME}

Let us focus on DO. Like PD verbs, DO verbs can take a non-pronominal theme (11) or a pronominal theme (10).

(10) John gave his ex wife all his money.
THEME

(11) John gave her all his money.
THEME

We want to know which DO verbs prefer pronominal themes and which prefer non-pronominal themes. We use the `dative` dataset from Harald Baayen's `languageR` package (Baayen 2009).

```
> rm(list=ls(all=TRUE))
> install.packages("languageR")
> library(languageR)
> data(dative)
```

The data set contains 3263 observations consisting of 15 variables. The variables divide into:

- speaker, a categorical variable with 424 levels, including NAs;
- modality, a categorical variable with two levels: spoken *vs.* written;
- verb, a categorical variable with 75 levels: *e.g. accord*, *afford*, *give*, *etc.*;
- semantic class, a categorical variable with five levels: abstract (*e.g. give* in *give it some thought*), transfer of possession (*e.g. send*), future transfer of possession (*e.g. owe*), prevention of possession (*e.g. deny*), and communication (*e.g. tell*);
- length in words of recipient, an integer valued variable;
- animacy of recipient, a categorical variable with two levels: animate *vs.* inanimate;
- definiteness of recipient, a categorical variable with two levels: definite *vs.* indefinite;
- pronominality of recipient, a categorical variable with two levels: pronominal *vs.* nonpronominal;
- length in words of theme, an integer valued variable;
- animacy of theme, a categorical variable with two levels: animate *vs.* inanimate;
- definiteness of theme, a categorical variable with two levels: definite *vs.* indefinite;
- pronominality of theme, a categorical variable with two levels: pronominal *vs.* nonpronominal;
- realization of recipient, a categorical variable with two levels: PD *vs.* DO;
- accessibility of recipient, a categorical variable with three levels: accessible, given, new;
- accessibility of theme, a categorical variable with three levels: accessible, given, new.

First, we restrict the data frame to those observations that illustrate DO (`dative$Realization OfRecipient == "NP"`)

```
> DO <- dative[which(dative$RealizationOfRecipient=="NP"),]; str(DO)
'data.frame': 2414 obs. of  15 variables:
 $ Speaker               : Factor w/ 424 levels "S0","S1000","S1001",..: NA NA NA NA NA NA NA
 NA NA NA ...
 $ Modality              : Factor w/ 2 levels "spoken","written": 2 2 2 2 2 2 2 2 2 2 ...
 $ Verb                  : Factor w/ 75 levels "accord","afford",..: 23 29 29 29 42 29 44 12 68
 29 ...
 $ SemanticClass         : Factor w/ 5 levels "a","c","f","p",..: 5 1 1 1 2 1 5 1 1 1 ...
 $ LengthOfRecipient     : int  1 2 1 1 2 2 2 1 1 1 ...
 $ AnimacyOfRec          : Factor w/ 2 levels "animate","inanimate": 1 1 1 1 1 1 1 1 1 1 ...
 $ DefinOfRec            : Factor w/ 2 levels "definite","indefinite": 1 1 1 1 1 1 1 1 1 1 ...
 $ PronomOfRec           : Factor w/ 2 levels "nonpronominal",..: 2 1 1 2 1 1 1 2 2 2 ...
 $ LengthOfTheme         : int  14 3 13 5 3 4 4 1 11 2 ...
 $ AnimacyOfTheme        : Factor w/ 2 levels "animate","inanimate": 2 2 2 2 2 2 2 2 2 2 ...
 $ DefinOfTheme          : Factor w/ 2 levels "definite","indefinite": 2 2 1 2 1 2 2 2 2 2 ...
 $ PronomOfTheme         : Factor w/ 2 levels "nonpronominal",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ RealizationOfRecipient: Factor w/ 2 levels "NP","PP": 1 1 1 1 1 1 1 1 1 1 ...
 $ AccessOfRec           : Factor w/ 3 levels "accessible","given",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ AccessOfTheme         : Factor w/ 3 levels "accessible","given",..: 3 3 3 3 3 3 3 3 1 1 ...
```

Next, we restrict the resulting data frame to two variables: `Verb` and `PronomOfTheme`.

```
> verb.theme <- DO[,c(3,12)]; str(verb.theme)
'data.frame': 2414 obs. of  2 variables:
 $ Verb         : Factor w/ 75 levels "accord","afford",..: 23 29 29 29 42 29 44 12 68 29 ...
 $ PronomOfTheme: Factor w/ 2 levels "nonpronominal",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Like above, we create an ordered frequency list with `count()` and then `arrange()` from the `plyr` package.

```
> library(plyr)
> freq <- count(DO, c("Verb","PronomOfTheme"))
> sorted.freq <- arrange(freq, PronomOfTheme, desc(freq)); head(sorted.freq, 15)
     Verb PronomOfTheme freq
1    give nonpronominal 1379
2    cost nonpronominal  158
3     pay nonpronominal  121
4    send nonpronominal   95
5    tell nonpronominal   62
6   offer nonpronominal   50
7   teach nonpronominal   49
8  charge nonpronominal   41
9    show nonpronominal   40
10     do nonpronominal   30
11  bring nonpronominal   25
12   sell nonpronominal   24
13    owe nonpronominal   23
14   feed nonpronominal   14
15  allow nonpronominal   12
```

By far, *give* is the DO verb that occurs the most with a non-pronominal theme. The same verb also occurs with pronominal themes, but not as much as *tell*.

```
> sorted.freq[which(sorted.freq$PronomOfTheme=="pronominal"),]
     Verb PronomOfTheme freq
53   tell    pronominal   60
54   give    pronominal   31
55  teach    pronominal   12
56   cost    pronominal   11
57   sell    pronominal   10
58   show    pronominal    9
59    owe    pronominal    6
60   send    pronominal    6
61    pay    pronominal    4
62  allow    pronominal    1
63    bet    pronominal    1
64 charge    pronominal    1
65   feed    pronominal    1
66 promise   pronominal    1
67  serve    pronominal    1
```

All in all, the DO construction prefers nonpronominal themes, based on raw frequencies.

9.2.3 Collostruction

In the Construction Grammar framework, Stefanowitsch and Gries have developed a family of methods known as collostructional analysis. Collexeme analysis measures the mutual attraction of lexemes and constructions (Stefanowitsch and Gries 2003). Distinctive collexeme analysis contrasts alternating constructions in their respective collocational preferences (Gries and Stefanowitsch 2004b). To measure the association between the two slots of the same construction, a third method known as covarying-collexeme analysis is used (Gries and Stefanowitsch 2004a; Stefanowitsch and Gries 2005).

9.3 Association Measures

Association measures measure the attraction or repulsion between two cooccurring linguistic phenomena. Although the basis of association measures, raw/observed frequency cannot serve to measure association by itself.

9.3.1 Measuring Significant Co-occurrences

Association measures are designed to capture statistically significant attractions or repulsions between two units. Raw/observed frequencies are not suited for the purpose of measuring significant associations. They should only serve as the basis for association measures, not for making conclusions in this regard.

Suppose you are comparing two words in five different contexts. The frequency distribution of both words is summarized in Tab. 9.1.

Table 9.1: A fictitious contingency table

	context 1	context 2	context 3	context 4	context 5
word 1	23	10	173	361	10
word 2	4	1	6	87	1

Comparing the contexts where the two words occur, we see that the observed frequency of word 1 in context 4 is higher than the observed frequency of word 2 in the same context. We might be tempted to conclude that word 1 prefers context 4. This is simply not the case. Let us see why.

First, we gather the results in a matrix.

```
> rm(list=ls(all=TRUE))
> r1 <- c(23, 10, 173, 361, 10)
> r2 <- c(4, 1, 6, 87, 1)
> matrix <- matrix(c(r1,r2), nrow=2, ncol=5, byrow=TRUE); matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]   23   10  173  361   10
[2,]    4    1    6   87    1
```

Next, we run a χ^2 test on the matrix and display the expected frequency of each cell.[2]

```
> chisq.test <- chisq.test(matrix)
> round(chisq.test$expected, 0)
     [,1] [,2] [,3] [,4] [,5]
[1,]   23    9  153  382    9
[2,]    4    2   26   66    2
```

The expected frequency of word 1 in context 4 is 382, i.e. larger than the observed frequency (361). This means that word 1 disprefers context 4. Expected frequencies are computed on the basis of the marginal

[2] R issues a warning because the conditions for the χ^2 test are not met, strictly speaking. Indeed, the frequencies are smaller than 5 in three cells.

totals of the table. If you interpret observed frequencies regardless of the marginal totals, you are running the risk of jumping to the wrong conclusion. Admittedly, there are fewer instances of word 2 in context 4 than there are instances of word 1 in the same context, but there are fewer instances of word 2 than word 1 in the whole table. Marginal totals allow you to take this into account.

Proportions in the form of percentages or normalized frequencies (such as the number of tokens per n words) are often proposed to account for distributional differences and allow for a fair comparison. Suppose you want to compare the distribution of a linguistic unit in two corpus samples, CORP1 and CORP2. CORP1 consists of 7543 words and CORP2 consists of 12714 words. All other things being equal, you are very likely to find more instances of the linguistic unit in CORP2 because it is larger. Suppose you find a count of 124 in CORP1, and 210 in CORP2. The frequencies normed to a basis per 10000 words is obtained by dividing the verb count by the number of words in the sample and multiplying the result by 10000.

```
> count1 <- 124 # number of instances of the linguistic unit in corpus 1
> count2 <- 210 # number of instances of the linguistic unit in corpus 2
> corp1 <- 75055 # size of corpus 1
> corp2 <- 127140 # size of corpus 2
> basis <- 10000 # normalization basis
> norm.freq1 <- (count1/corp1)*basis
> round(norm.freq1,2) # normalized frequency
[1] 16.52
> norm.freq2 <- (count2/corp2)*basis
> norm.freq2 <- round(norm.freq2,2); norm.freq2
[1] 16.52
```

The normalized frequencies tell you that the distributions of the linguistic unit across the two samples are similar. However, normalized frequencies do not tell you whether the observed distribution is significantly different from chance.

9.3.2 The Logic of Association Measures

Most association measures are computed on the basis of a contingency table such as Tab. 9.2.

Table 9.2: An input contingency table for the measure of association between two words (¬: "other than")

	W_2	$\neg W_2$	row totals
W_1	a	b	a+b
$\neg W_1$	c	d	c+d
column totals	a+c	b+d	a+b+c+d

Cell a denotes the number of times W_1 and W_2 co-occur, cell b the number of times W_1 co-occurs with words other than W_2, cell c the number of times W_2 co-occurs with words other than W_1, and cell d the number of times that words other than W_1 and W_2 co-occur.

The point of Tab. 9.2 is to compare the observed occurrences of two forms with their expected frequencies to determine which collocations are noteworthy, i.e. statistically significant. In this respect, the important parts are the row totals, the column totals, and the table totals, as you know from Sect. 9.3.1.

9.3.3 A Quick Inventory of Association Measures

The inventory of association measures is vast. They are extensively reviewed in Church et al. (1991), Evert (2005, 2009), and Pecina (2010).[3] The sections below will guide you through four popular association measures: mutual information, Fisher's exact test, χ^2, and log-likelihood ratio.

9.3.3.1 Mutual Information

A product of information theory, pointwise MI determines to what extent the occurrences of a word W_1 influence the occurrences of another word W_2. Because it is intuitively and methodologically simple, British lexicographers have made pointwise MI a standard association measure (see Kennedy 2003).

Church and Hanks (1990, p. 23) define $I(W_1, W_2)$, the mutual information I of two words W_1 and W_2, as follows:

$$I(W_1, W_2) = log_2 \frac{P(W_1, W_2)}{P(W_1)P(W_2)}, \tag{9.1}$$

where $P(W_1)$ and $P(W_2)$ are the respective probabilities of W_1 and W_2, $P(W_1, W_2)$ is the probability of observing W_1 and W_2 together, and $P(W_1)(W_2)$ the probability of observing W_1 and W_2 independently, i.e. by chance.[4] The underlying assumption is that W_1 and W_2 are independent. $I(W_1, W_2) = 0$ if and only if W_1 and W_2 are independent.

$P(W_1)$ and $P(W_2)$ are obtained by counting the occurrences of W_1 and W_2 in the corpus and normalizing the frequency by the corpus size. Take the compound noun *computer scientist* in the Corpus of Contemporary American English (Davies 2008–2012).

Let W_1 denote *computer* and W_2 denote *scientist*. The compound noun occurs 316 times in the corpus. The corpus contains 533788932 words (as of July 4, 2016). We compute $P(W_1, W_2)$ by dividing 316 by the corpus size.

```
> P_W1_W2 <- 316/533788932
> P_W1_W2
[1] 5.919943e-07
```

There are 63135 instances of *computer* and 12068 occurrences of *scientist*. We compute $P(W_1)$ and $P(W_2)$ likewise.

```
> P_W1 <- 63135/533788932
> P_W1
[1] 0.0001182771
> P_W2 <- 12068/533788932
> P_W2
[1] 2.260819e-05
```

All we need now is to compute $P(W_1)P(W_2)$.

```
> P_W1_P_W2 <- P_W1*P_W2
> P_W1_P_W2
[1] 2.674031e-09
```

[3] See also Stefan Evert's comprehensive inventory of existing association measures: http://www.collocations.de/AM/ (last accessed on July 1, 2016).

[4] The symbol log_2 means "the logarithm to the base 2". The binary logarithm is the favored expression of the amount of self-information and information entropy in information theory.

We may now calculate $I(W_1, W_2)$.

```
> I <- log2(P_W1_W2/P_W1_P_W2)
> I
[1] 7.790424
```

Although the hypothesis of independence is rejected if $I(W_1, W_2) > 0$, Church and Hanks (1990, p. 24) have observed that the most interesting pairs are observed when $I(W_1, W_2) > 3$. Here, $I = 7.79$, which indicates a significant attraction between *computer* and *scientist*.

Mutual information is sensitive to data sparseness and invalid for low-frequency word pairs (Evert 2009; Kilgarriff 2001; Manning and Schütze 1999). Consequently, pointwise MI favors low-frequency collocates (Church et al. 1991, p. 133). I do not recommend that you use it, unless you are familiar with more recent versions of mutual information such as MI^2 or MI^3 (not covered in this book).

9.3.3.2 Fisher's Exact Test

Fisher's exact test, which we know from Sect. 8.10, can be used to measure the association between two lexemes. The `fisher.test()` function takes as input a matrix such as Tab. 9.3.

Table 9.3: An input contingency table for the measure of association between *computer* and *scientist* in COCA

	scientist	¬ *scientist*
computer	316	62819
¬ *computer*	11752	533714045

First, we make the matrix,

```
> m <- matrix(c(316, 62819, 11752, 533714045), nrow=2, ncol=2, byrow=TRUE)
```

and then we run the `fisher.test()` function specifying `alternative="greater"`. Like mutual information, Fisher's exact test assumes the null hypothesis that the two words are independent from each other. We are hoping to see a p-value that is as close to zero as possible so as to reject the null hypothesis.

```
> fisher.test(m, alternative = "greater")

Fisher's Exact Test for Count Data

data:  m
p-value < 2.2e-16
alternative hypothesis: true odds ratio is greater than 1
95 percent confidence interval:
 206.5396      Inf
sample estimates:
odds ratio
  228.9237
```

Because the p-value is very small, we reject the null hypothesis of independence. The association between *computer* and *scientist* is beyond doubt.

The test computes the odds ratio to quantify the association between the presence or absence of W_1 and the presence or absence of W_2. The null hypothesis of independence is equivalent to the hypothesis that the

odds ratio equals 1. The alternative for a one-sided test is based on the odds ratio, so `alternative = "greater"` is a test of the odds ratio being greater than 1.

In a 2×2 contingency table with joint distributions such as Tab. 9.3, the odds of success within the first row are

$$\Omega_{computer} = \frac{P(computer, scientist)}{P(\neg computer, scientist)}, \tag{9.2}$$

and the odds of success within the second row are

$$\Omega_{\neg computer} = \frac{P(scientist, \neg computer)}{P(\neg computer, \neg scientist)}. \tag{9.3}$$

The odds ratio is the ratio of the odds $\Omega_{computer}$ and $\Omega_{\neg computer}$:

$$\theta = \frac{\Omega_{computer}}{\Omega_{\neg computer}}. \tag{9.4}$$

If $\theta = 1$, then W_1 and W_2 are independent. The further θ is from 1, the stronger the association (Agresti 2002, pp. 44–45).

```
> omega_computer <- (m[1,1]/sum(m))/(m[2,1]/sum(m))
> omega_not_computer <- (m[1,2]/sum(m))/(m[2,2]/sum(m))
> odds.ratio <- omega_computer/omega_not_computer
> odds.ratio
[1] 228.4509
```

Here, $\theta = 228.45$, indicating that we have a strong association between *computer* and *scientist*. Because we have cell counts, we can compute what is known as the sample odds ratio $\widehat{\theta}$, which is:

$$\widehat{\theta} = \frac{n_{computer\ scientist}\ n_{\neg computer\ \neg scientist}}{n_{\neg computer\ scientist}\ n_{\neg computer\ scientist}}. \tag{9.5}$$

```
> sample.OR <- (m[1,1]*m[2,2])/(m[1,2]*m[2,1]); sample.OR
[1] 228.4509
```

We obtain the same result. Note that the θ score we obtain (228.45) differs very slightly from what `fisher.test()` returns (228.92). This is because the function estimates the odds ratio via conditional maximum likelihood estimate instead of the unconditional maximum likelihood estimate (i.e. the sample odds ratio).

Because this association measure is symmetric, it does not change if you transpose the table and declare *scientist* as the response variable instead of *computer*.

```
> fisher.test(t(m), alternative="greater")

Fisher's Exact Test for Count Data

data:  t(m)
p-value < 2.2e-16
alternative hypothesis: true odds ratio is greater than 1
95 percent confidence interval:
 206.5396      Inf
sample estimates:
odds ratio
  228.9237
```

9.3.3.3 The χ^2 Test

We normally use χ^2 to test the null hypothesis that two lexemes are independent. We pit H_0 against the alternative hypothesis that they are associated. We can also use χ^2 to measure the association between two lexemes. Using Tab. 9.2 as a reference point, χ^2 is defined as follows:

$$\chi^2 = \sum_{ij} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}, \tag{9.6}$$

where O_{ij} is the observed value for the cell in row i and column j, and E_{ij} is the expected value under the null hypothesis. Once applied to Tab. 9.2, formula (9.6) becomes:

$$\begin{aligned} \chi^2 = & \left(\frac{(O_{11} - E_{11})^2}{E_{11}}\right) \\ & + \left(\frac{(O_{12} - E_{12})^2}{E_{12}}\right) \\ & + \left(\frac{(O_{21} - E_{21})^2}{E_{21}}\right) \\ & + \left(\frac{(O_{22} - E_{22})^2}{E_{22}}\right). \end{aligned} \tag{9.7}$$

The expected frequency for a cell in row i and column j of a contingency table is found using the formula:

$$E_{ij} = \frac{N_i N_j}{N}, \tag{9.8}$$

where E_{ij} denotes the expected frequency of cell i, j, N_i is the sum total of the ith row, N_j the sum total of the jth column, and N the sum total of the contingency table. First, we compute a matrix `m.exp` of expected frequencies.

```
> N_row1 <- rowSums(m)[1] # sum total of row 1
> N_row2 <- rowSums(m)[2] # sum total of row 2
> N_col1 <- colSums(m)[1] # sum total of column 1
> N_col2 <- colSums(m)[2] # sum total of column 2
> N <- sum(m) # sum total of the contingency table
> E11 <- (N_row1*N_col1/N) # expected freq of cell in row 1 col 1
> E12 <- (N_row1*N_col2/N) # expected freq of cell in row 1 col 2
> E21 <- (N_row2*N_col1/N) # expected freq of cell in row 2 col 1
> E22 <- (N_row2*N_col2/N) # expected freq of cell in row 2 col 2
> m.exp <- matrix(c(E11, E12, E21, E22), nrow=2, ncol=2, byrow=TRUE)
> m.exp
             [,1]         [,2]
[1,]     1.427368     63133.57
[2,] 12066.572632 533713730.43
```

Next, we compute the χ^2 score using the indices of the cells in `m` and `m.exp`.

```
> chi <- ((m[1,1]-m.exp[1,1])^2/m.exp[1,1]) + ((m[1,2]-m.exp[1,2])^2/m.exp[1,2]) +
+   ((m[2,1]-m.exp[2,1])^2/m.exp[2,1]) + ((m[2,2]-m.exp[2,2])^2/m.exp[2,2])
> chi
[1] 69337.33
```

The χ^2 value is 69337.88. Looking up the χ^2 distribution table in Sect. A.2.3, we find that the critical value of χ^2 is 3.8415 when there is one degree of freedom (because we have a 2×2 table) and α is set to 0.05. We cannot reject the null hypothesis that *computer* and *scientist* are independent.

Of course, a faster solution is to use the `chisq.test()` function from base R.

```
> chisq.test(m)$statistic
X-squared
 69117.09
```

The score is different from what we obtain when we do the calculation manually. This is because the `chisq.test()` function applies Yate's continuity correction by default (enter `?chisq.test()` to know more about the correction).[5] To override the default setting, specify `correct=FALSE`, and you obtain the same χ^2 score as above.

```
> chisq.test(m, correct=FALSE)$statistic
X-squared
 69337.33
```

9.3.3.4 The Log-Likelihood Ratio Test

The log-likelihood ratio (Dunning 1993), abbreviated G^2, is often used as an alternative to the χ^2 test (see below). It is preferred when the samples are small.[6] According to Stefan Evert,[7] G^2 gives an excellent approximation of the exact p-values of the Fisher exact association measure.

Like the tests above, G^2 is used to test two textual hypotheses:

H_0: the probability of observing W_2 given W_1 is the same as the probability of observing W_2 when W_1 is absent;
H_1: the probability of observing W_2 given W_1 is different from the probability of observing W_2 when W_1 is absent.

Once operationalized, these translate into the following:

H_0: $P(W_2|W_1) = P(W_2|\neg W_1)$;
H_1: $P(W_2|W_1) \neq P(W_2|\neg W_1)$.

For a contingency table with i rows and j columns, G^2 is defined as follows:

$$G^2 = 2\sum_i O_{ij} log(\frac{O_{ij}}{E_{ij}}), \tag{9.9}$$

where O_{ij} stands for an observed count at row i and column j and E_{ij} an expected count at row i and column j. Thanks to the matrix of expected frequencies that we made in the previous section (`m_exp`), we can now apply formula (9.9) to Tab. 9.3:

[5] It is assumed that the binomial frequencies observed in a contingency table follow a continuous χ^2 distribution. Stricly speaking, this gap may introduce some error. The continuity correction is meant to correct the error. It substracts 0.5 from the absolute value of all $O-E$ differences. It is generally applied when the conditions of validity of the test are not met (for example when one cell count is smaller than 5). Consequently, it reduces the χ^2 score and increases its corresponding p-value.

[6] http://www.collocations.de/AM/section4.html.

[7] http://www.collocations.de/AM/section4.html#s4.3.

$$\begin{aligned} G^2 = & 2((316 \times log(\frac{316}{1.427368}))+ \\ & (62819 \times log(\frac{62819}{63133.57}))+ \\ & (11752 \times log(\frac{11752}{12066.572632}))+ \\ & (533714045 \times log(\frac{533714045}{533713730.43}))). \end{aligned} \tag{9.10}$$

In R, we compute G^2 as follows.

```
> G2 <- 2*(m[1,1]*log(m[1,1]/m.exp[1,1]) +
+            m[1,2]*log(m[1,2]/m.exp[1,2]) +
+            m[2,1]*log(m[2,1]/m.exp[2,1]) +
+            m[2,2]*log(m[2,2]/m.exp[2,2]))
> G2
[1] 2793.441
```

According to Manning and Schütze (1999, p. 172), the G^2 statistic is more interpretable than χ^2 scores because the user does not need to use a table of critical values. In practice, however, because G^2 has an asymptotic χ^2 distribution, we can use the χ^2 distribution table to assess the significance of the association (Kotze and Gokhale 1980; Read and Cressie 1988; Cressie and Read 1989). A G^2 score of 3.8415 or higher is significant at the level of $p < 0.05$, and a score of 10.8276 is significant at the level of $p < 0.001$. The magnitude of the G^2 score that we obtain (2793.441) leaves no doubt as to the significance of the association between *computer* and *scientist*. We can safely reject H_0.

9.3.4 A Loop for Association Measures

In the examples above, the computations are based on a single contingency table. Suppose that, like most corpus linguists, you work with a large data set such as `computer.NP.rds` in your `chap9` folder. The data set ranks nouns based on how often they co-occur with *computer* in COCA. Tab. 9.4 provides an overview of the data set.

For each collocate of *computer*, you need to make a contingency table such as Tab. 9.2. To make the table, you use the frequency information provided in the data set and do the maths to obtain the missing cells. Let us see how it works for the first line of the data set (*screen*). You start from four numbers: the corpus size (533788932), the frequency of *computer scientist* in the corpus (1633), the frequency of *computer* in the corpus (63135), and the frequency of *screen* in the corpus (36510). The data set gives you two of these numbers. The corpus size and the frequency of *computer* can be obtained by respectively looking at the corpus documentation and by making a simple corpus query. Tab. 9.5 shows where to insert these figures in a contingency table.

You now have everything you need to fill in the blanks. As a rule, start with the easy calculations (cells *b* and *c* in Tab. 9.2) and finish with the cell at the intersection of ¬*computer* and ¬*screen* (cell *d* in Tab. 9.2). Tab. 9.6 shows one way of filling in the blanks.

We want a script that computes the association between *computer* and all the collocates in the table using χ^2. First, we load the data.

```
> rm(list=ls(all=T))
> data <- readRDS("C:/CLSR/chap9/computer.NP.rds") # Windows
```

Table 9.4: An input data set for the measure of the association between *computer* and its right nominal collocates in COCA

rank	W2	freq_W_1_W_2	corp_freq_W_2
1	*screen*	1633	36510
2	*science*	1375	80786
3	*systems*	1145	63837
4	*system*	1023	192311
5	*software*	895	30765
6	*program*	861	168218
7	*technology*	754	79997
8	*games*	616	65210
9	*programs*	610	93327
10	*models*	569	37398
11	*industry*	504	76517
12	*networks*	401	17283
13	*simulations*	381	2215
14	*programmer*	372	1130
15	*lab*	369	17851
...	...	...	...
3507	*states*	1	246428
3508	*american*	1	278556
3509	*president*	1	318222
3510	*us*	1	522829

Table 9.5: An input contingency table to measure the association between *computer* and *screen* in COCA

	screen	¬ *screen*	row totals
computer	1633		63135
¬ *computer*			
column totals	36510		533788932

Table 9.6: Filling in the blanks in Tab. 9.2

	screen	¬ *screen*	row totals
computer	1633	63135 – 1633	63135
¬ *computer*	36510 – 1633	(533788932 – 63135) – (36510 – 1633)	533788932 – 63135
column totals	36510	533788932 – 36510	533788932

```
> data <- readRDS("/CLSR/chap9/computer.NP.rds") # Mac
```

Next, we vectorize some fixed values, namely the frequency of W_1 in the corpus and the corpus size.

```
> freq_W1 <- 63135
> corp.size <- 533788932
```

Since we are going to loop over the `data` data frame, we create an empty character vector to collect all the χ^2 scores.

```
> all.chi <- character()
```

It is time to enter the loop. For each W_2 (i.e. each row in the data frame), we want to:

- compute the a, b, c, and d cells of the input contingency table;
- combine these values into a 2×2 matrix;
- compute a matrix of expected frequencies under the null hypothesis of independence;
- calculate χ^2 for each contingency table.

This is how we do it.

```
> for (i in 1:nrow(data)){ # enter the loop
+  a <- data$freq_W1_W2[i] # cell a
+  b <- freq_W1 - a # cell b
+  c <- data$corp_freq_W2[i] - a # cell c
+  d <- (corp.size - freq_W1) - c # cell d
+  m <- matrix(c(a, b, c, d), nrow=2, ncol=2, byrow=TRUE) # make the 2x2 matrix
+  N <- sum(m) # sum total of the contingency table
+  N_row1 <- rowSums(m)[1] # sum total of row 1
+  N_row2 <- rowSums(m)[2] # sum total of row 2
+  N_col1 <- colSums(m)[1] # sum total of column 1
+  N_col2 <- colSums(m)[2] # sum total of column 2
+  E11 <- (N_row1*N_col1/N) # expected freq of cell in row 1 col 1
+  E12 <- (N_row1*N_col2/N) # expected freq of cell in row 1 col 2
+  E21 <- (N_row2*N_col1/N) # expected freq of cell in row 2 col 1
+  E22 <- (N_row2*N_col2/N) # expected freq of cell in row 2 col 2
+  m.exp <- matrix(c(E11, E12, E21, E22), nrow=2, ncol=2, byrow=TRUE) # matrix of expected freqs
+  current.chi <- ((m[1,1]-m.exp[1,1])^2/m.exp[1,1]) +
+    ((m[1,2]-m.exp[1,2])^2/m.exp[1,2]) +
+    ((m[2,1]-m.exp[2,1])^2/m.exp[2,1]) +
+    ((m[2,2]-m.exp[2,2])^2/m.exp[2,2])
+   current.chi <- round(current.chi, 2) # round the score
+   current.chi <- paste(data$W2[i], current.chi, sep="\t") # append word 2
+  all.chi <- c(all.chi, current.chi) # collect all scores
+ } # exit the loop
```

Upon inspection, we see that the script has done its job.

```
> head(all.chi)
[1] "screen\t614385.64"   "science\t195177.04"  "systems\t171393.56"  "system\t44007.2"
[5] "software\t218386.77" "program\t35572.49"
```

We save `all.chi` in a text file so that it can be read with a spreadsheet software.

```
> cat(all.chi, sep="\n", file="/CLSR/chap9/chisq.scores.txt")
```

Tab. 9.7 is a snapshot of what you see when you sort the data table according to the χ^2 score with the spreadsheet software.[8] For large data sets, manual calculations are tedious and unproductive. The point of the above script is to use it as a basis for the computation of other association measures.

[8] Strangely, the corpus frequencies of W_2 that you obtain with individual corpus queries via http://corpus.byu.edu/coca differ slightly from those that you obtain in bulk. This is a bug, which might alter your χ^2 scores minimally.

Table 9.7: Top 10 collocates of *computer* in COCA

W_2	χ^2 score
screen	614385.64
software	218386.77
science	195177.04
systems	171393.56
models	72074.01
technology	58602.49
games	47985.15
system	44007.2
program	35572.49
programs	32510.02

9.3.5 There Is No Perfect Association Measure

Parametric tests select a statistical test so that its distribution converges to a well-known distribution when the null hypothesis is true. These tests are powerful and easy to compute, but they suppose that the data is similar to one of these known distributions. Since co-occurrence data do not always satisfy these distributional assumptions, exact tests may be more appropriate. Exact tests compute the p-value of all possible outcomes that are similar to or greater than the observed frequencies of a contingency table.

Neither parametric tests nor exact tests are without problems. The χ^2 test presupposes that the linguistic phenomenon under scrutiny is distributed randomly across a corpus, but "language is never, ever, ever, random" (Kilgarriff 2005). Fisher's exact test (Yates 1984) is considered more appropriate than parametric tests in the identification of dependent word pairs because it is not affected by the skewed and sparse nature of data samples typical of natural-language corpora (Pedersen 1996). In contrast to the above tests, mutual information does not depend on distributional assumptions and can be computed at minimal computational cost.

All things considered, no association measure is fully satisfactory, and the choice of one over another depends on what aspects of collocativity one wishes to reveal (Evert 2005; Wiechmann 2008). For all the above, one should always interpret ranked lists with caution.

9.3.6 Collostructions

In the context of lexico-grammatical patterns, Stefanowitsch and Gries have developed a family of three methods known as collostructional analysis. These methods are: collexeme analysis (Stefanowitsch and Gries 2003), distinctive collexeme analysis (Gries and Stefanowitsch 2004b), and co-varying collexeme analysis (Gries and Stefanowitsch 2004a; Stefanowitsch and Gries 2005).

Collostructional analysis starts from contingency tables and applies an association measure (Tab. 9.8). It is highly consonant with the lexicometric traditions of Britain (see Sect. 1.2.3.1) and France (Lafon 1980, 1981, 1984; Lebart et al. 1998; Muller 1964, 1973, 1977).

Table 9.8: Association measures involved in collostructional analysis

methods	association measures
collexeme analysis	Fisher's exact test, odds ratio, G^2, MI, χ^2
distinctive collexeme analysis	Fisher's exact test, G^2
distinctive collexeme analysis (3+ alternants)	multinomial test
co-varying collexeme analysis	Fisher's exact test, odds ratio, G^2

The nice thing about collostructional analysis is that Stefan Gries has written an interactive R script: `coll.analysis.r` (Gries 2014a). The script guides you throughout the process. All you need to do is download it once from Stefan Gries's homepage.

```
> source("http://www.linguistics.ucsb.edu/faculty/stgries/teaching/groningen/coll.analysis.r")
```

In the sections that follow, I briefly describe what each method does. Hilpert (2014) and to the collostructional analysis instruction page (`http://www.linguistics.ucsb.edu/faculty/stgries/teaching/groningen/readme.txt`) show how the data sets should be formatted and how additional data should be entered.

9.3.6.1 Collexeme Analysis

Collexeme analysis measures the mutual attraction of lexemes and constructions. This requires that Tab. 9.2 be formatted so that it accommodates the construction as one of its two variables. In Tab. 9.9, the construction is the response variable.

Table 9.9: Input contingency table for collexeme analysis (L: lexeme, C: construction)

	L_j	$\neg L_j$	row totals
C_i	a	b	a+b
$\neg C_i$	c	d	c+d
column totals	a+c	b+d	a+b+c+d

Tab. 9.10 is a sample of the input table that I used in a study of *quite* (among three other adverbs) when it intensifies adjectives (Desagulier 2014, p. 159). One of the purposes of the study was to see what adjectives were significantly associated with each intensifier. The data frame consists of three columns: the adjectives, the frequencies of the adjectives in the corpus, and the frequencies of the adjectives in the *quite* + *ADJ* construction.

Tab. 9.10 was submitted to `coll.analysis()`. G^2 was chosen as an association measure. Tab. 9.11 was output. Here is what the column headers stand for:

- `word.freq`: the frequency of each adjective in the corpus;
- `obs.freq`: the observed frequency of each adjective in <*quite* + A>;
- `exp.freq`: the expected frequency of each adjective in <*quite* + A>;
- `faith`: the percentage of how many instances of the adjective occur in <*quite* + A>;

Table 9.10: Input data set for collexeme analysis (*quite* + A)

adjective	frequency of adjective in corpus	frequency of adjective in construction
different	17012	2247
sure	137372	1347
clear	81553	805
good	378826	578
right	4558	548
possible	88919	458
similar	60967	328
ready	55338	300
common	63239	278
simple	48134	271
...	...	...

- `relation`: the relation of the adjective to *quite* (attraction vs. repulsion);
- `delta.p.constr.to.word`: ΔP, i.e. how much the construction <*quite* + A> helps guess the adjective;
- `delta.p.word.to.constr`: ΔP, i.e. how much the adjective helps guess the construction <*quite* + A>;
- `coll.strength`: the index of collocational/collostructional strength based on G^2.

Table 9.11: Output table for collexeme analysis (*quite* + A)

words	word.freq	obs.freq	exp.freq	relation	faith	delta.p.constr.to.word	delta.p.word.to.constr	coll.strength
different	170210	2247	29.849509	attraction	0.013201	0.030487	0.013031	15082.42588
sure	137372	1347	24.090751	attraction	0.009805	0.018191	0.009633	8231.3208
clear	81553	805	14.301845	attraction	0.009871	0.010872	0.009697	4923.956471
possible	88919	458	15.593611	attraction	0.005151	0.006083	0.004976	2216.176977
similar	60967	328	10.691705	attraction	0.00538	0.004363	0.005205	1614.270513
good	378826	578	66.43423	attraction	0.001526	0.007034	0.001352	1482.015768
ready	55338	300	9.704554	attraction	0.005421	0.003992	0.005247	1480.807683
simple	48134	271	8.441198	attraction	0.00563	0.00361	0.005455	1357.461688
remarkable	12161	186	2.132659	attraction	0.015295	0.002528	0.01512	1297.761456
common	63239	278	11.090142	attraction	0.004396	0.00367	0.004221	1259.479394
...	...	...	...	...	...	...	...	...

The data frame `input.ca.rds` compares the raw frequencies and the G^2 scores. The code below generates Fig. 9.1, which helps you visualize this comparison.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap9/input.ca.rds") # Windows
> data <- readRDS("/CLSR/chap9/input.ca.rds") # Mac
> plot(data$obs.freq~data$coll.strength, xlab="log-likelihood ratio", ylab="observed frequency",
+     col="white", type="n")
> text(data$obs.freq~data$coll.strength, labels=row.names(data), cex=0.8)
```

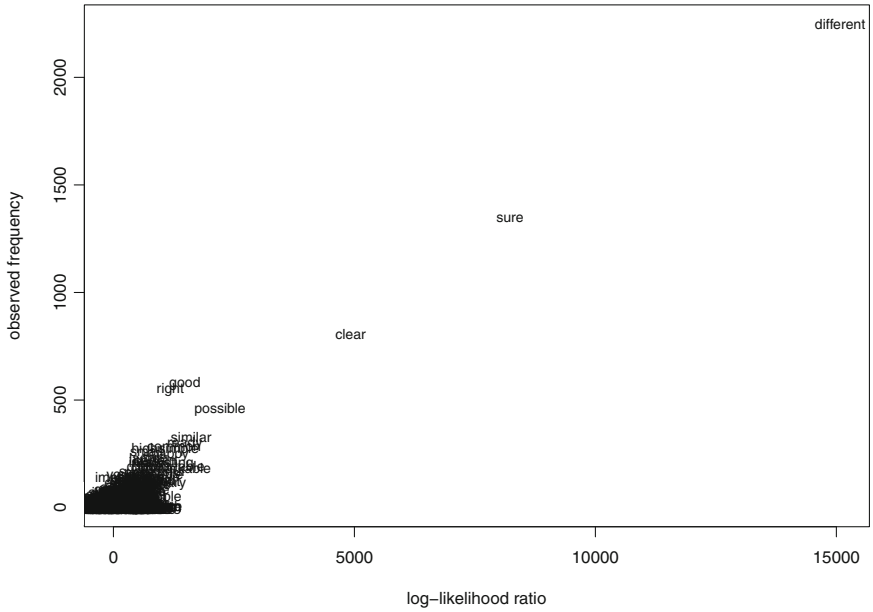


Fig. 9.1: A comparison of observed frequency and G^2 scores in collexeme analysis

Although both measures are strongly correlated for the adjectives that have both high observed frequencies and high G^2 scores,

```
> cor(data$obs.freq, data$coll.strength)
[1] 0.9713854
```

we see that such items as *good* or *right* are not as attracted to <*quite* + A> as their observed frequencies suggest.

9.3.6.2 Distinctive Collexeme Analysis

Distinctive collexeme analysis (henceforth DCA) contrasts alternating constructions in their respective collocational preferences. Tab. 9.12 is the kind of contingency table that is used as input in DCA.

Table 9.12: Input contingency table for DCA

	L_j	$\neg L_j$	row totals
C_1	a	b	a+b
C_2	c	d	c+d
column totals	a+c	b+d	a+b+c+d

Gries and Stefanowitsch (2004b) use DCA to show which verbs exhibit a strong preference for which variant of the dative alternation in the ICE-GB corpus. They find that the most distinctive verb of DO is *give*, along with verbs that express a transaction (literal or figurative) involving a direct contact between agent and recipient (*tell*, *teach*, *ask*, *show*, *offer*, *promise*, *deny*). The most distinctive verb of PD is *bring*, along with verbs involving some distance between agent and recipient (*take*, *pass*, *make*, *sell*, *do*, etc.).

Let us replicate Gries and Stefanowitsch's experiment on the `dative` data set from Harald Baayen's `languageR` package. Tab. 9.9 is what the input data set for `coll.analysis()` looks like. For each row, the script computes a contingency table such as Tab. 9.12.

You obtain Tab. 9.13 from the original data set by selecting the two variables of interest: `RealizationOfRecipient`, and `Verb`.

```
> rm(list=ls(all=TRUE))
> library(languageR)
> data(dative)
> data <- dative[, c(13, 3)]
> head(data)
  RealizationOfRecipient  Verb
1                     NP  feed
2                     NP  give
3                     NP  give
4                     NP  give
5                     NP offer
6                     NP  give
```

The resulting data frame is available from the `chap9` folder (`input.dca.rds`).

Table 9.13: Input data set for DCA

construction	verb
DO	*give*
PD	*send*
DO	*teach*
PD	*sell*
DO	*give*
DO	*give*
DO	*tell*
PD	*take*
DO	*give*
DO	*give*
...	...

Once you have submitted the data frame to DCA with `coll.analysis()` and followed the required steps, you obtain Tab. 9.14.[9] Here is what the column headers stand for:

- `obs.freq.double.object`: the observed frequency of the verb in the double object construction;
- `obs.freq.prep.dative`: the observed frequency of the verb in the prepositional dative construction;
- `exp.freq.double.object`: the expected frequency of the verb in the double object construction;
- `exp.freq.prep.dative`: the expected frequency of the verb in the preposition dative construction;
- `pref.occur`: the construction to which the verb is attracted;

[9] Again, the collostructional strength is based on G^2.

- `delta.p.constr.to.word`: ΔP, i.e. how much the construction helps guess the verb;
- `delta.p.word.to.constr: delta p`: ΔP, i.e. how much the verb helps guess the construction;
- `coll.strength`: index of distinctive collostructional strength based on G^2.

To compare PD and DO more easily, I have reformatted the table so as to display the distinctive collexemes of each construction side by side (Tab. 9.15).

The data frame `pd.do.rds` compares the raw frequencies and the G^2 scores. The code below generates Fig. 9.2, which helps you visualize this comparison.

Table 9.14: Output table for DCA

verbs	pref.const	obs.freq.double.object	obs.freq.prep.dative	exp.freq.double.object	exp.freq.prep.dative	delta.p.constr.to.word	delta.p.word.to.constr	coll.strength
give	DO	1410	256	1232.5234	433.4766	0.2826	0.2177	204.6726
cost	DO	169	0	125.0279	43.9721	0.07	0.2744	105.0698
tell	DO	122	6	94.6957	33.3043	0.0435	0.222	42.4776
teach	DO	61	3	47.3478	16.6522	0.0217	0.2176	20.9271
charge	DO	42	1	31.8118	11.1882	0.0162	0.2401	18.6752
…	…	…	…	…	…	…	…	…
make	PD	3	3	4.4389	1.5611	-0.0023	-0.2403	1.5718
hand	PD	9	6	11.0971	3.9029	-0.0033	-0.1405	1.3973
extend	PD	2	2	2.9592	1.0408	-0.0015	-0.2401	1.0471
grant	PD	9	4	9.6175	3.3825	-0.001	-0.0477	0.1476
allot	PD	2	1	2.2194	0.7806	-3.00E-04	-0.0732	0.0791

Table 9.15: Top 10 collexemes distinguishing between PD and DO based on the `dative` dataset in `languageR`

PD		DO	
collexemes	distinctiveness	collexemes	distinctiveness
sell	323.7443	*give*	204.6726
take	137.3388	*cost*	105.0698
bring	20.4579	*tell*	42.4776
send	20.0388	*teach*	20.9271
pay	19.5323	*charge*	18.6752
issue	16.1876	*do*	12.0191
loan	12.0077	*owe*	8.092
write	11.2308	*allow*	7.8537
lease	10.7847	*wish*	5.4333
award	8.6362	*show*	3.7647

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap9/pd.do.rds") # Windows
> data <- readRDS("/CLSR/chap9/pd.do.rds") # Mac
> plot(data$obs.freq~data$coll.strength, xlab="log-likelihood ratio", ylab="observed frequency",
+      type="n")
> text(data.total$obs.freq~data.total$coll.strength, labels=row.names(data), cex=0.8)
```

The conclusions of Gries and Stefanowitsch (2004b) are partly verified. In particular, the profiles *give* and *sell* stand out. Interestingly, *give* has the highest observed frequency in the table, but not the highest

G^2 score. Conversely, *sell* does not have the highest observed frequency in the table, but the highest G^2 score. This time, the linear relationship between observed frequencies and collostructional strength is far from perfect.

```
> cor(data$obs.freq, data$coll.strength)
[1] 0.5986189
```

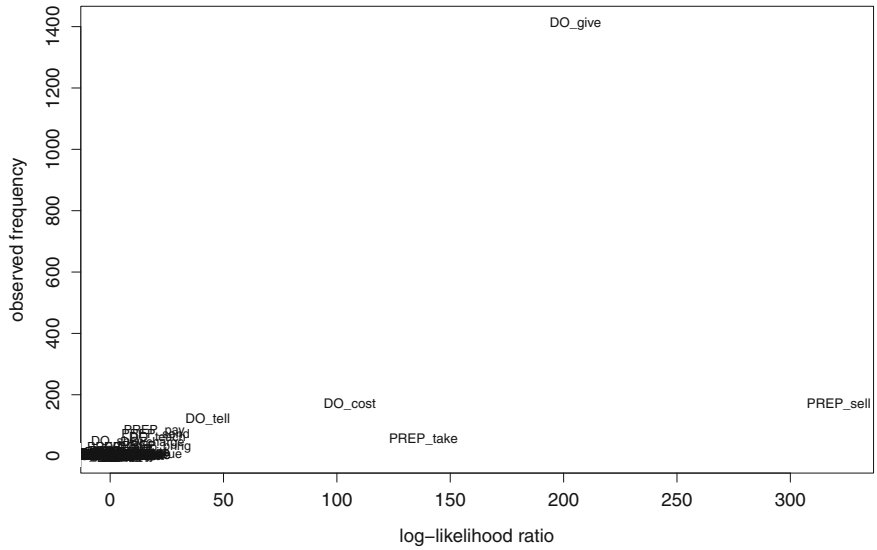


Fig. 9.2: A comparison of observed frequency and G^2 scores in DCA

9.3.6.3 Covarying Collexeme Analysis

Finally, co-varying collexeme analysis (henceforth CCA) measures the association between the two lexical slots of the same construction. Tab. 9.16 is the kind of contingency table that is used as input in CCA.

Table 9.16: Input contingency table for CCA

	$L_{slot\ 1}$	$\neg L_{slot\ 1}$	row totals
$L_{slot\ 2}$	a	b	a+b
$\neg L_{slot\ 2}$	c	d	c+d
column totals	a+c	b+d	a+b+c+d

In Desagulier (2016), I use CCA using G^2 as an association metric to measure the association between the lexemes that instantiate the adjective and NP positions in *A as NP* in the BNC, e.g. *right as rain*, *cool as a cucumber*, etc. (Desagulier 2016). CCA determines for each potential adjective occurring in the first slot

which potential NPs in the second slot co-occur with it more often than expected. Tab. 9.17 is a sample of the input table that I submitted to CCA.

Table 9.17: Input data set for CCA

A	NP
big	*football*
safe	*houses*
large	*life*
steady	*rock*
sure	*hell*
bold	*brass*
old	*hills*
gentle	*lamb*
silent	*grave*
bright	*sun*
...	...

Once you have submitted the data frame to DCA with `coll.analysis()` and followed the required steps, you obtain Tab. 9.18. Here is what the column headers stand for:

- `words1`: words in the adjective slot of *A as NP*;
- `words2`: words in the NP slot of *A as NP*;
- `freq.w1`: frequency of the adjective in *A as NP*;
- `freq.w2`: frequency of the NP in *A as NP*;
- `obs.w1_2.in_c`: observed frequency of both A and NP in both slots of *A as NP*;
- `exp.w1_2.in_c`: expected frequency of both A and NP in both slots of *A as NP*;
- `relation`: relation between observed and expected frequency;
- `coll.strength`: index of co-varying collexeme strength: G^2.

Table 9.18: Output table for CCA

words1	words2	freq.w1	freq.w2	obs.w1_2.in_c	exp.w1_2.in_c	relation	coll.strength
good	*gold*	29	30	29	0.48	attraction	288.8138
quick	*flash*	27	20	20	0.3	attraction	189.2885
right	*rain*	20	20	18	0.22	attraction	175.9832
large	*life*	18	21	17	0.21	attraction	164.5468
safe	*houses*	17	14	14	0.13	attraction	148.3236
sure	*hell*	50	98	33	2.69	attraction	141.9763
old	*hills*	31	15	15	0.26	attraction	130.8729
pretty	*picture*	16	12	12	0.11	attraction	126.4332
bold	*brass*	12	10	10	0.07	attraction	113.2006
solid	*rock*	22	26	15	0.31	attraction	110.9541
...	...	...	...	...	...	...	...
sharp	*hell*	36	98	1	1.94	repulsion	0.5895
hard	*hell*	23	98	1	1.24	repulsion	0.0527

The data frame `a_as_np.rds` compares the observed frequencies and the G^2 scores. The code below generates Fig. 9.3, which helps you visualize this comparison. This time, we plot a regression line in red thanks to the last line of code.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap9/a_as_np.rds") # Windows
> data <- readRDS("/CLSR/chap9/a_as_np.rds") # Mac
> plot(data$obs.w1_2.in_c~data$coll.strength, xlab="log-likelihood ratio",
+      ylab="observed frequency", col="white", type="n")
> text(data$obs.w1_2.in_c~data$coll.strength, labels=row.names(data), cex=0.8)
> abline(lm(data$obs.w1_2.in.c~data$coll.strength), col="red", lwd=2) # plot a regression line
```

The two measures are strongly correlated.

```
> cor(data$obs.w1_2.in_c, data$coll.strength)
[1] 0.9117879
```

Nevertheless, some profiles stand out, especially *good as gold* and *sure as hell*. This is made visible by their respective positions vis-à-vis the regression line. To a lesser extent, the profiles of *cold as ice*, *white as snow*, *solid as a rock*, and *white as a sheet* also stand out because their observed frequencies do not allow to predict their degrees of association accurately.

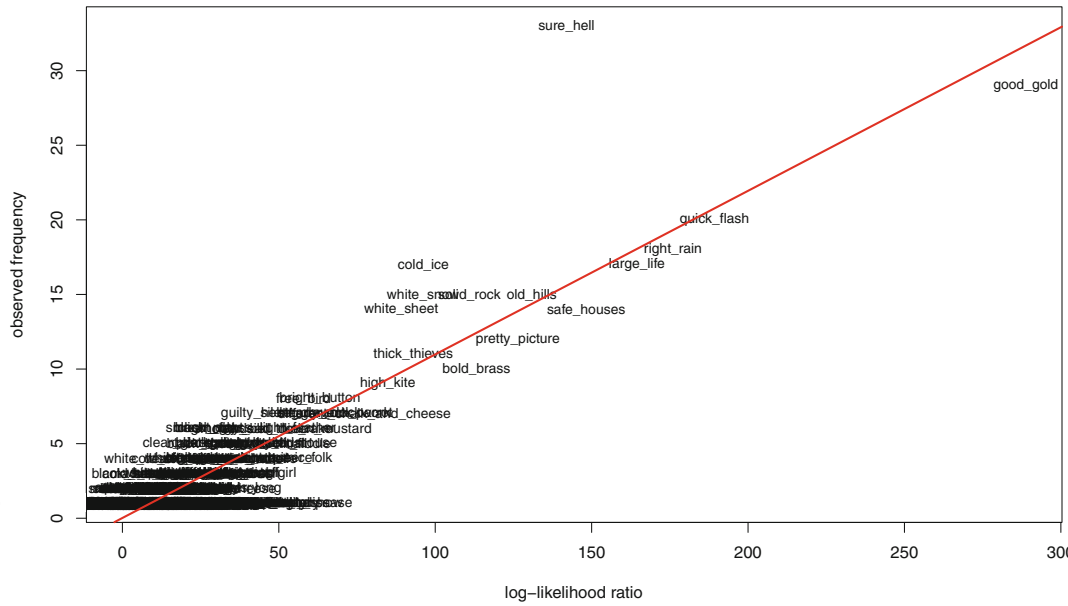


Fig. 9.3: A comparison of observed frequency and G^2 scores in CCA

9.3.7 Asymmetric Association Measures

9.3.7.1 Associative Learning

All the association measures that we have seen above posit a bidirectional dependency between the collocates. Therefore, given the pair $collocate_1$-$collocate_2$, $collocate_1$ attracts or repels $collocate_2$ as much as $collocate_2$ attracts or repels $collocate_1$.[10]

Yet, given *facto*, the probability of obtaining *ipso* is high, but given *facto*, the probability of obtaining *ipso* is not as high because of other words compete for the same slot (e.g. *de* or *post*). In other words, the collocation between *ipso* and *facto* is directional because one word is a better cue of the other than vice versa.

The idea that collocations are directional has been put forth by Gries (2013). It is inspired by studies on language acquisition which draw from a rejection of classical conditioning in the field of associative learning.[11] In classical conditioning, learning occurs when a "cue" (or conditioned stimulus) is temporally paired with an "outcome" (or unconditioned stimulus). Pavlov (1927) presents a cue (ringing a bell) to a dog just before an outcome (food). After several identical experiments, the dog emits the conditional reflex of salivating upon being presented with the cue alone. Pavlov concludes that the outcome reinforces the conditioned reflex of salivating to the cue. According to classical conditioning, it is the temporal pairing of the cue and the outcome that associates the two and therefore strengthens the conditioned reflex. Rescorla (1968) contends that rats do not emit a conditioned reflex when trials involving the outcome alone are added to trials where the temporal pairing between cue and the outcome is preserved. He concludes that it is in fact contingency, not temporal pairing, that generates the conditioned response.

According to Ellis (2006) and Baayen (2011), Rescorla's revision of Pavlovian conditioning is at the root of contemporary inferential methods in animal and human learning. In line with Bayesian reasoning, the Rescorla-Wagner equations state that learners predict an outcome from the cues available in their environment if such cues have a value in terms of outcome prediction, information gain, and statistical association (Wagner and Rescorla 1972). Drawing a parallel with the automatic completion feature of smartphones, whose algorithms take into account recency of prior usage, frequency of prior usage, and context of usage to perform automatic completion, it is now assumed that the association between a cue and an outcome is directional.

9.3.7.2 Asymmetric Collocations

When an event comprises a cue and an outcome, it can summarized in a contingency table such as Tab. 9.19.

Table 9.19: Contingency table involving a cue (C) and an outcome (O)

	O	¬O
C	a	b
¬C	c	d

[10] To verify this, we have seen that transposing the contingency table with `t()` did not change the score of the association measure.

[11] In particular Ellis (2006); Ellis and Ferreira Junior (2009).

To measure the asymmetric dependency between C and O, Ellis relies on ΔP, a one-way-dependency statistic developed by Allan (1980):

$$\begin{aligned} \Delta P &= P(O|C) - P(O|\neg C) \\ &= \frac{a}{a+b} - \frac{c}{c+d} \end{aligned} \tag{9.11}$$

The closer ΔP is to 1, the more C increases the likelihood of O. Conversely, the closer ΔP is to -1, the more C decreases the likelihood of O. If $\Delta P = 0$, there is no covariation between C and O.

Gries (2013) has transposed the above to the study of bigrams. Given a bigram involving two words, W_1 and W_2, the frequencies can be gathered in a contingency table such as Tab. 9.20.

Table 9.20: Contingency table involving a cue (W_1) and an outcome (here, W_2)

	W_2: present	W_2: absent
W_1: present	a	b
W_1: absent	c	d

Two ΔP values must be computed, depending on whether the cue is W_1 and the outcome is W_2 or whether the cue is W_2 or and the outcome is W_1:

$$\begin{aligned} \Delta P_{(W_2|W_1)} &= P(W_2|W_1) - P(W_2|\neg W_1) \\ &= \frac{a}{a+b} - \frac{c}{c+d} \end{aligned} \tag{9.12}$$

$$\begin{aligned} \Delta P_{(W_1|W_2)} &= P(W_1|W_2) - P(W_1|\neg W_2) \\ &= \frac{a}{a+c} - \frac{c}{b+d} \end{aligned} \tag{9.13}$$

If $\Delta P_{(W_2|W_1)} - \Delta P_{(W_1|W_2)}$ is positive, then W_1 is a better predictor of W_2 than vice versa. Conversely, if $\Delta P_{(W_2|W_1)} - \Delta P_{(W_1|W_2)}$ is negative, then W_2 is a better predictor of W_1 than vice versa. If $\Delta P_{(W_2|W_1)} - \Delta P_{(W_1|W_2)}$ is null, then no word is a good predictor of the other.

Let us apply formulas (9.12) and (9.13) to *computer screen*. Let *computer* be the cue (W_1) and *screen* the outcome (W_2). We obtain Tab. 9.21.

Table 9.21: A contingency table for *computer screen* in COCA

	screen: present	*screen*: absent
computer: present	1633	61502
computer: absent	34877	533690920

On the basis of the contingency table, we compute the two ΔP values as follows:

$$\begin{aligned}\Delta P_{(W_{screen}|W_{computer})} &= P(W_{screen}|W_{computer}) - P(W_{screen}|\neg W_{computer}) \\ &= \frac{a}{a+b} - \frac{c}{c+d} \\ &= \frac{1633}{1633+61502} - \frac{34877}{34877+533690920},\end{aligned} \tag{9.14}$$

$$\begin{aligned}\Delta P_{(W_{computer}|W_{screen})} &= P(W_{computer}|W_{screen}) - P(W_{computer}|\neg W_{screen}) \\ &= \frac{a}{a+c} - \frac{c}{b+d} \\ &= \frac{1633}{1633+34877} - \frac{34877}{61502+533690920}.\end{aligned} \tag{9.15}$$

We now calculate the two values in R.

```
> m <- matrix(c(1633, 61502, 34877, 533690920), nrow=2, ncol=2, byrow=TRUE)
> delta.W2.W1 <- (m[1,1]/(m[1,1]+m[1,2])) - (m[2,1]/(m[2,1]+m[2,2]))
> delta.W1.W2 <- (m[1,1]/(m[1,1]+m[2,1])) - (m[2,1]/(m[1,2]+m[2,2]))
> delta.W2.W1 - delta.W1.W2
[1] -0.01886227
```

Because $\Delta P_{(W_2|W_1)} - \Delta P_{(W_1|W_2)}$ is negative, W_2 is a better predictor of W_1 than vice versa. This is easy to understand. Although both words are paired with many other words than each other, Tab. 9.21 shows that *screen* occurs in fewer word pairs ($1633+34877=36510$) than *computer* ($1633+61502=63135$). What ΔP does is quantify this asymmetry. Although negative, the ΔP difference is very close to 0, suggesting a near equilibrium. Comparatively, there is a much bigger asymmetry between *computer* and *programmer* ($\Delta P_{diff} = -0.3233$), suggesting that *programmer* is a much better predictor of *computer* than *screen* is.

Below is a script that computes all the ΔP differences in `computer.NP.rds`.

```
> rm(list=ls(all=T))
> data <- readRDS("C:/CLSR/chap9/computer.NP.rds") # Windows
> data <- readRDS("/CLSR/chap9/computer.NP.rds") # Mac
> str(data)
> freq_W1 <- 63135 # frequency of computer in corpus
> corp.size <- 533788932
> all.delta.diff <- character() # empty character vector to collect the results
> for (i in 1:nrow(data)){ # enter the loop
+  a <- data$freq_W1_W2[i] # cell a
+  b <- freq_W1 - a # cell b
+  c <- data$corp_freq_W2[i] - a # cell c
+  d <- (corp.size - freq_W1) - c # cell d
+  m <- matrix(c(a, b, c, d), nrow=2, ncol=2, byrow=TRUE) # make the 2x2 matrix
+  current.delta.W2.W1 <- (m[1,1]/(m[1,1]+m[1,2])) - (m[2,1]/(m[2,1]+m[2,2])) # first delta
+  current.delta.W1.W2 <- (m[1,1]/(m[1,1]+m[2,1])) - (m[2,1]/(m[1,2]+m[2,2])) # second delta
+  current.delta.diff <- current.delta.W2.W1 - current.delta.W1.W2 # delta difference
+  current.delta.diff <- round(current.delta.diff, 4) # round
+    current.delta.diff <- paste(data$W2[i], current.delta.diff, sep="\t") # append word 2
+  all.delta.diff <- c(all.delta.diff, current.delta.diff) # collect all delta differences
+ } # exit the loop
```

The results are vectorized in `all.delta.diff`.

```
> head(all.delta.diff, 15)
 [1] "screen\t-0.0189"     "science\t0.0048"     "systems\t2e-04"      "system\t0.0109"
 [5] "software\t-0.0149"   "program\t0.0085"     "technology\t0.0025"  "games\t3e-04"
 [9] "programs\t0.0031"    "models\t-0.0062"     "industry\t0.0014"    "networks\t-0.0169"
[13] "simulations\t-0.166" "programmer\t-0.3233" "lab\t-0.0148"
```

9.3.7.3 Asymmetric Collostructions

In Desagulier (2016), I transposed Tab. 9.20 and formulas (9.12) and (9.13) to the study of multiple-slot constructions, and more specifically *A as NP*. The input for the measure ΔP is a contingency table (Tab. 9.22) that is similar to those that we have used so far. The choice of W_{slot1} as the cue and W_{slot2} as the outcome is arbitrary.

Table 9.22: Contingency table involving a cue (W_{slot1}) and an outcome (W_{slot_2})

	W_{slot_2}: present	W_{slot_2}: absent
W_{slot1}: present	a	b
W_{slot1}: absent	c	d

On the basis of the contingency table, we compute the two ΔP values like above:

$$\begin{aligned}\Delta P_{(W_{slot2}|W_{slot1})} &= P(W_{slot2}|W_{slot1}) - P(W_{slot2}|\neg W_{slot1}) \\ &= \frac{a}{a+b} - \frac{c}{c+d}\end{aligned} \tag{9.16}$$

$$\begin{aligned}\Delta P_{(W_{slot1}|W_{slot2})} &= P(W_{slot1}|W_{slot2}) - P(W_{slot1}|\neg W_{slot2}) \\ &= \frac{a}{a+c} - \frac{c}{b+d}\end{aligned} \tag{9.17}$$

If $\Delta P_{(W_{slot2}|W_{slot1})} - \Delta P_{(W_{slot1}|W_{slot2})}$ is positive, then W_{slot1} is a better predictor of W_{slot2} than vice versa. Conversely, if the difference is negative, then W_{slot2} is a better predictor of W_{slot1} than vice versa. If the difference is null, then neither W_{slot1} nor W_{slot2} is a good predictor. For the sake of illustration, let us now apply Tab. 9.22 and formulas (9.16) and (9.17) to one concrete example of *A as NP*: *mad as a March hare*. We obtain Tab. 9.23 and formulas (9.18) and (9.19):

Table 9.23: Co-occurrence table for *mad* and *March hare* in *mad as a March hare* in the BNC

	March hare: present	*March hare*: absent
mad: present	2	7
mad: absent	0	98363774

$$\begin{aligned}\Delta P_{(March\ hare|mad)} &= P(March\ hare|mad) - P(March\ hare|\neg mad) \\ &= \frac{2}{2+7} - \frac{0}{0+98363774} \\ &\approx 0.22\end{aligned} \tag{9.18}$$

$$\begin{aligned}\Delta P_{(mad|March\ hare)} &= P(mad|March\ hare) - P(mad|\neg March\ hare)\\ &= \frac{2}{2+0} - \frac{0}{7+98363774}\\ &= 1\end{aligned} \tag{9.19}$$

$$\Delta P_{(March\ hare|mad)} - \Delta P_{(mad|March\ hare)} \approx -0.78 \tag{9.20}$$

Unsurprisingly, *March hare* is a much better predictor of *mad* than vice versa. This is because *mad* co-occurs with other NPs such as *hatter* or *hell*. On the other hand, *March hare* co-occurs exclusively with *mad* in the BNC. In other words, because *mad* is far more productive than *March hare* in *A as NP*, the former is much less of a good predictor than the other in the corpus. All in all, asymmetric association measures tell us something that symmetric association measures do not.

9.4 Lexical Richness and Productivity

In Desagulier (2016), I investigate whether association measures have some bearing on the study of productivity. They do, to some extent. However, association measures do not replace the range of established measures of lexical richness and productivity.

Historically, the reflection on the quantification of linguistic productivity stems from morphology. Most of the techniques that we use in lexicology are therefore influenced by works on morphological productivity. Plag (2003) provides a theoretical definition of morphological productivity. It is the property of an affix or a word-formation rule to generate new units systematically. The more an affix is available to form new words, the more productive it is. In practice, just because a coinage is rare does not mean that it should not be accounted for Bauer (2001). At a quantitative level, measures have been developed to capture the productivity of morphological processes in corpora. For reasons of space, I can present but the tip of the iceberg. In the sections below, I apply the measures of lexical richness and productivity to the lexicon.

9.4.1 Hapax-Based Measures

Baayen (1989, 1993) and Baayen and Lieber (1991) *inter alia* have developed a set of measures based on the idea that the number of hapax legomena (words that occur only once in a corpus) of a given morphological category is correlated with the number of neologisms in that category, and therefore with the productivity of the rule at work.

$\mathscr{P}^*$ measures "expanding productivity", that is to say the "hapax-conditioned degree of morphological productivity"'. It is the number of hapax legomena in the category $V(1,C,N)$ divided by the number of hapax legomena in the corpus $V(1,N)$:

$$\mathscr{P}^* = \frac{V(1,C,N)}{V(1,N)}. \tag{9.21}$$

$\mathscr{P}^*$ does not account for whether a productive morphological process is saturated and is not used that much. On the other hand, $\mathscr{P}$ measures "potential productivity" by calculating the probability of encountering new types. It is the ratio of the number of hapax legomena with a given affix and the sum of all tokens that contain the affix:

$$\mathscr{P} = \frac{V(1,C,N)}{N(C)}. \tag{9.22}$$

The larger $\mathscr{P}$, the larger the number of hapax legomena and the productivity of the affix. Conversely, the larger the sum of all tokens, the smaller $\mathscr{P}$ and the productivity of the affix. One mathematical property of $\mathscr{P}$ is that it is the slope of the tangent to a vocabulary growth curve at its endpoint. It indicates the rate at which vocabulary increases at the end of a vocabulary growth curve (see below). Baayen (1993) explores further the correlation between productivity and frequency and observes that $\mathscr{P}$ captures only the probability of possible types to the detriment of observed types. He advocates a measure of global productivity (P^*) which evaluates $\mathscr{P}$ in terms of type frequency (i.e. actual use):

$$P^* = \frac{V}{\mathscr{P}}. \tag{9.23}$$

In Desagulier (2016), I found 217 adjectives that appear only once in *A as NP* for a total of 402 types. I found 665 NPs that appear only once in the construction for a total of 877 types. There are 1819 instances of *A as NP* in the BNC. These figures serve as input for the computation of $\mathscr{P}$ and P^* as follows.

```
> P_potential_adj <- 217/1819
> P_potential_adj
[1] 0.1192963
> P_potential_NP <- 665/1819
> P_potential_NP
[1] 0.3655855
> P_global_adj <- 402/P_potential_adj
> P_global_adj
[1] 3369.76
> P_global_NP <- 877/P_potential_NP
> P_global_NP
[1] 2398.892
```

In the construction, the NP slot is more productive than the adjective slot in terms of potential productivity. This is due to a greater variety in the choice of NPs. This variety is strengthened by the fact that NPs vary with respect to the singular/plural agreement, the determiner, and optional pre- and post-modifiers. Comparatively, the formal variability of adjectives is more limited. Yet, the potential for intrinsic variability of NPs remains to be seen beyond the data set, as shown by the lower P^* score when compared to the score of adjectives. Potentially, fewer adjectives than NPs can appear in the first slot of the construction, yet more adjectives types than NP types are observed.

The decisive feature of hapax-based measures is that they show how complex productivity is. Depending on the measure that you use, you reveal a specific facet of the productivity of a linguistic phenomenon.

9.4.2 Types, Tokens, and Type-Token Ratio

Type frequency is perhaps the most basic productivity measure. In Sect. 5.4, we saw that the sentence *The cat is on the mat* contains six word tokens but five word types. In this sentence the token frequency is the total number of words (6). The type frequency is the total number of unique words (5). Type is commonly abbreviated V (for vocabulary) or $V(C,N)$, i.e. the type count of the members of a linguistic category C in a corpus of N tokens. $V(C,N)$ measures what Baayen (2009) calls "realized productivity".

Type-token ratio (henceforth TTR) is a very popular measure of lexical richness. It is obtained by dividing the number of types in a corpus by the number of tokens. A high TTR (close to 1) suggest that the corpus is characterized by lexical diversity.[12] A low TTR (close to 0) suggests that there are lexical repetitions.

Although you can calculate TTR manually from the frequency lists that you have learned to make in Chap. 4,[13] several text-mining packages have functions for the calculation of TTR. One that is simple to use is `type_token_ratio()` from the `qdap` package.

```
> install.packages("qdap")
> library("qdap")
```

First, vectorize the text file whose TTR you want, namely the Declaration of Independence, from `chap9`.

```
> text <- scan("C:/CLSR/chap9/doi.txt", what="char", sep="\n") # Windows
> text <- scan("/CLSR/chap9/doi.txt", what="char", sep="\n") # Mac
```

Next, run the `type_token_ratio()` function.

```
> type_token_ratio(text.var = text)
  all   wc   ttr
1 all 1321 0.475

Type-token ratio for entire text: 0.475
```

The score suggests that the text is not so productive because there are repetitions.

TTR should be handled with care because it fails to account for two facts:

- the distribution of types across a corpus is not homogeneous;
- the larger your corpus, the harder it becomes to find new types as you progress into the text.

Consequently, TTR cannot be used to compare the productivity of two corpora of different sizes because you do not know how types are distributed in each corpus, and the larger corpus may end up having a lower TTR because of the increasing repetition of grammatical words.

9.4.3 Vocabulary Growth Curves

One solution to the abovementioned issue is to compute the standardized TTR for each corpus. Each corpus is split into equal chunks and the TTR is calculated for each chunk. The mean TTR of all chunks gives you the standardized TTR. In a similar vein, Baayen (1993, 2001, 2008) suggests that V be plotted against the number of tokens at multiple intervals to keep track of vocabulary development across a corpus. The result is a vocabulary growth curve (henceforth VGC). The VGC in Fig. 9.4 displays the productivity of Cervantes's *Don Quijote* at the level of V (types) and $V1$ (hapax legomena).[14]

Underlying Baayen's hapax-based measures is the following principle: productivity is a factor of both a large number of low-frequency words and a low number of high-frequency words. Zipf (1949) showed that even the largest collection of text in a given language does not contain instances of all types in that language. Therefore, type count and distribution in a corpus cannot reliably estimate type count and distribution in a language.

[12] If TTR = 1, this means that the number of types equals the number of token. In other words, all the word tokens are different.

[13] For a given frequency list with r rows, simply divide r by the sum of the token frequencies. That will give you the TTR of the file from which the frequency list was made.

[14] It was obtained with the code in Baayen (2008, Sect. 6.5).

To circumvent this issue and predict the vocabulary size in a larger corpus based on a much smaller corpus, Baayen (2001) uses LNRE models (the acronym stands for "large number of rare events"). First, one converts the data into a frequency spectrum (i.e. a table of frequencies of frequencies). Next, one fits a model to the spectrum object. Once a satisfactory LNRE model has been fitted to the frequency spectrum, one obtains expected values for the vocabulary size and the spectrum elements within the range of the corpus/sample size (interpolation) and beyond (extrapolation). One can then plot the results on an enhanced VGC. In Fig. 9.5, the VGC is extrapolated to twice the size of the novel.

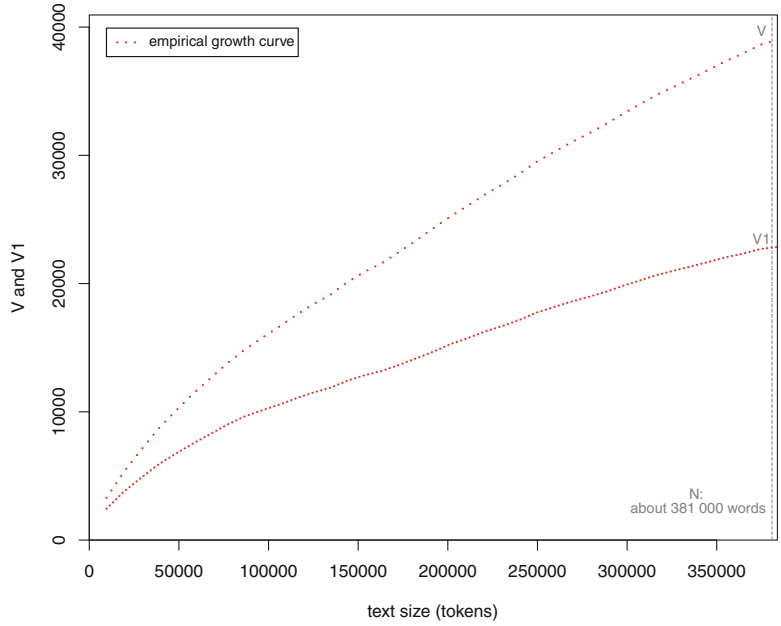


Fig. 9.4: Empirical VGCs of *Don Quijote*

Zeldes (2012) shows that Baayen's family of hapax-based measures and LNRE models apply equally well to morphosyntactic constructions. In Desagulier (2016), I show the relevance of plotting VGCs in the productivity assessment of *A as NP*. Let us compare the productivity of each slot of the construction.

The data is in `prod.a.as.np.rds` in the `chap9` folder.

```
> df <- readRDS("/CLSR/chap9/prod.a.as.np.rds") # Windows
> df <- readRDS("/CLSR/chap9/prod.a.as.np.rds") # Mac
```

We have just loaded a data frame. Normally, VGCs are plotted on the basis of a text. Here, we are working with two "texts": `ADJ` and `NP`, which we extract and vectorize.

```
> adj<-as.character(df$ADJ)
> np<-as.character(df$NP)
```

We are going to process each of them in turn, starting with adjectives. In the preamble, we load two packages.

```
> library(languageR)
> library(zipfR)
```

First, we use the `growth.fnc()` function from Harald Baayen's `languageR` package. The text is divided into 20 chunks (`nchunks = 20`), each of which consists of 90 tokens. For each chunk, a range of lexical measures is calculated.[15]

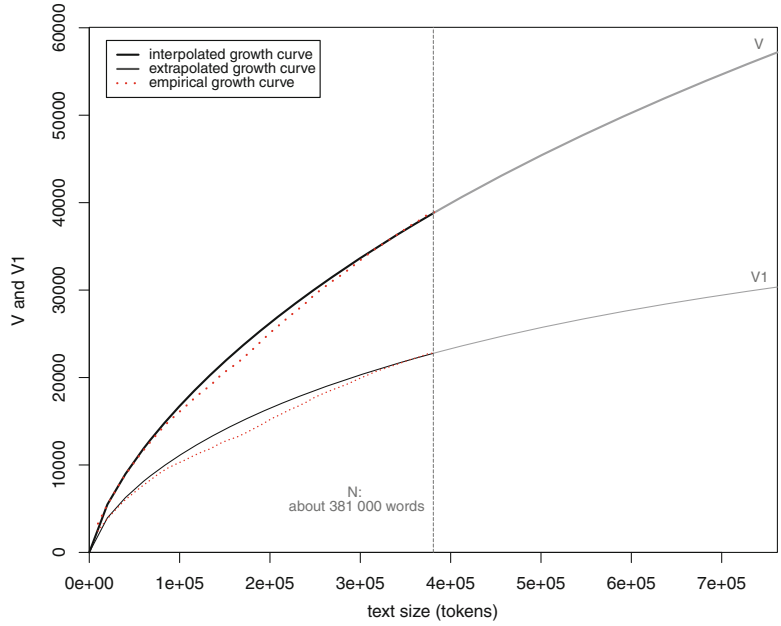


Fig. 9.5: Empirical, interpolated, and extrapolated VGCs of *Don Quijote*

```
> df.growth = growth.fnc(text = adj, size=90, nchunks = 20)
.....................
> str(df.growth)
Formal class 'growth' [package "languageR"] with 1 slot
  ..@ data:List of 1
  .. ..$ data:'data.frame': 20 obs. of  13 variables:
  .. .. ..$ Chunk         : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
  .. .. ..$ Tokens        : num [1:20] 90 180 270 360 450 540 630 720 810 900 ...
  .. .. ..$ Types         : num [1:20] 65 103 136 163 190 215 232 247 266 282 ...
  .. .. ..$ HapaxLegomena : num [1:20] 48 68 89 104 123 134 143 150 158 163 ...
  .. .. ..$ DisLegomena   : num [1:20] 12 17 18 17 15 23 24 26 29 37 ...
  .. .. ..$ TrisLegomena  : num [1:20] 3 7 9 16 16 17 18 22 24 20 ...
  .. .. ..$ Yule          : num [1:20] 91.4 103.7 102.1 98.9 94.9 ...
  .. .. ..$ Zipf          : num [1:20] -0.377 -0.468 -0.499 -0.564 -0.515 ...
  .. .. ..$ TypeTokenRatio: num [1:20] 0.722 0.572 0.504 0.453 0.422 ...
  .. .. ..$ Herdan        : num [1:20] 1.109 0.986 0.903 0.85 0.808 ...
  .. .. ..$ Guiraud       : num [1:20] 6.85 7.68 8.28 8.59 8.96 ...
  .. .. ..$ Sichel        : num [1:20] 0.1846 0.165 0.1324 0.1043 0.0789 ...
  .. .. ..$ Lognormal     : num [1:20] 0.225 0.362 0.418 0.473 0.497 ...
```

The function outputs a growth object which we will need when we plot the empirical growth curve.

```
> df.vgc.a = growth2vgc.fnc(df.growth)
```

[15] Yule's *K*, the Zipf slope, TTR, Herdan's *C*, Guiraud's *D*, Sichel's *S*, the mean of log frequency (Baayen 2008, p. 224).

Next, we convert the data into a frequency spectrum thanks to `spc()`, from the `zipfR` package. This is a two-step process.

```
> df.table = table(table(adj))
> df.spc = spc(m = as.numeric(names(df.table)), Vm = as.numeric(df.table))
```

Thanks to this spectrum, you obtain the number of types and the number of tokens.

```
> sum(df.spc$Vm) # number of types
[1] 402
> sum(df.spc$m * df.spc$Vm) # number of tokens
[1] 1819
```

It is time to fit an LNRE model with the `lnre()` function which takes the spectrum object as its second argument. The first argument is chosen from a range of LNRE models available in the `zipfR` package (Evert and Baroni 2006, 2007), namely the Zipf-Mandelbrot model (`zm`), the finite Zipf-Mandelbrot model (`fzm`), or the Generalized Inverse Gauss-Poisson model (`gigp`). The goodness of fit of the model to the empirical distribution of each configuration is evaluated with a multivariate χ^2 test (Baayen 2001, Sect. 3.3). According to Baayen (2008, p. 233), "[f]or a good fit, the χ^2-value should be low, and the corresponding p-value large and preferably well above 0.05". With respect to our data, it is the finite Zipf-Mandelbrot model that provides the best fit. The `lnre()` function fits the model to the frequency spectrum.

```
> df.lnre.fzm.a = lnre("fzm", df.spc)
> df.lnre.fzm.a
finite Zipf-Mandelbrot LNRE model.
Parameters:
   Shape:          alpha = 0.4980175
   Lower cutoff:       A = 7.445123e-09
   Upper cutoff:       B = 0.02975264
 [ Normalization:      C = 2.931993 ]
Population size: S = 65711.95
Sampling method: Poisson, with exact calculations.

Parameters estimated from sample of size N = 1819:
               V     V1   V2    V3   V4    V5
   Observed: 402 217.00 59.0 26.00 15.0 13.00 ...
   Expected: 402 216.64 54.6 27.33 17.1 11.98 ...

Goodness-of-fit (multivariate chi-squared test):
         X2 df         p
   7.038904  6 0.3172644
```

The model can therefore provide reliable interpolated and extrapolated expected values for the vocabulary size and the spectrum elements. Thanks to the `lnre.vgc()` function, we use the model to obtain expected values for the data at smaller sample sizes (this is know as interpolation), and at larger sample sizes (this is known as extrapolation).

```
> # interpolation
> df.int.fzm.a = lnre.vgc(df.lnre.fzm.a, seq(0, N(df.lnre.fzm.a), length = 20), m.max=3)
> # extrapolation
> df.ext.fzm.a = lnre.vgc(df.lnre.fzm.a, seq(N(df.lnre.fzm.a), N(df.lnre.fzm.a)*2,
+                                            length = 20), m.max = 3)
```

With respect to interpolation, the first argument of the function is the model. The second argument is a sequence from 0 to the "text" size (1819) with 20 intervals.[16] With respect to extrapolation, the second argument is a sequence from 0 to twice the "text" size (3638), because we want to see what would happens if we had twice as much data of the same kind.

[16] The third argument `m.max` includes VGCs for spectrum elements up to the selected integer.

We proceed likewise for NPs.

```
> df.growth = growth.fnc(text = np, size=90, nchunks = 20)
.....................
> df.vgc.np = growth2vgc.fnc(df.growth)
> df.table = table(table(np))
> df.spc = spc(m = as.numeric(names(df.table)), Vm = as.numeric(df.table))
> df.lnre.fzm.np = lnre("fzm", df.spc) # fzm provides the best fit
> df.lnre.fzm.np
finite Zipf-Mandelbrot LNRE model.
Parameters:
   Shape:          alpha = 0.7234165
   Lower cutoff:       A = 3.05174e-20
   Upper cutoff:       B = 0.01538211
 [ Normalization:      C = 0.8775525 ]
Population size: S = 1.591057e+14
Sampling method: Poisson, with exact calculations.

Parameters estimated from sample of size N = 1819:
               V     V1    V2    V3    V4    V5
   Observed: 877 665.00 81.00 39.00 20.00 16.00 ...
   Expected: 877 652.42 90.23 38.39 21.85 14.32 ...

Goodness-of-fit (multivariate chi-squared test):
         X2 df         p
   8.349324  6 0.2136115
> df.int.fzm.np = lnre.vgc(df.lnre.fzm.np, seq(0, N(df.lnre.fzm.np), length=20), m.max=3)
> df.ext.fzm.np = lnre.vgc(df.lnre.fzm.np, seq(N(df.lnre.fzm.np),
+                                              N(df.lnre.fzm.np)*2, length = 20), m.max = 3)
```

We may now plot the VGCs. The code below plots the empirical VGCs of *A* and *NP* in *A as NP* (Fig. 9.6).

```
> plot(df.vgc.np, df.vgc.a, add.m=1, col=c("red", "coral3"), lty=c(3,3),
+      lwd=c(3,3),  main="", ylab="")
> # add.m=1 allows to plot the VGC for hapax legomena
>
> ylab_name=expression(paste("V"," and"," V"[1],sep="")) # label of y axis
> title(ylab=ylab_name) # plot the label of y axis
>
> text(1750, 890, expression("V"[NP]), cex=0.8) # plot annotations
> text(1750, 680, expression("V1"[NP]), cex=0.8) # plot annotations
> text(1750, 430, expression("V"[A]), cex=0.8) # plot annotations
> text(1750, 240, expression("V1"[A]), cex=0.8) # plot annotations
>
> legend<-legend("topleft", inset=.025,c("NP slot (empirical)", "A slot (empirical)"),
+                lty=c(3,3), lwd=c(3,3), col=c("red", "coral3"),
+                bg="white", cex=0.7) # plot the legend
```

The code below plots the empirical and interpolated VGCs of *A* and *NP* in *A as NP* (Fig. 9.7).

```
> plot(df.int.fzm.np, df.vgc.np, df.int.fzm.a, df.vgc.a, add.m=1,
+      col=c("black","red", "black", "coral3"), lty=c(1,3,2,3),
+      lwd=c(2,2,2,2),  main="", ylab="")
>
> ylab_name=expression(paste("V"," and"," V"[1],sep=""))
> title(ylab=ylab_name)
>
> text(1750, 890, expression("V"[NP]), cex=0.8)
> text(1750, 680, expression("V1"[NP]), cex=0.8)
> text(1750, 430, expression("V"[A]), cex=0.8)
> text(1750, 240, expression("V1"[A]), cex=0.8)
>
> legend<-legend("topleft", inset=.025,c("NP slot (interpolated)", "NP slot (empirical)",
+                "A slot (interpolated)","A slot (empirical)"),
+                lty=c(1,3,2,3), lwd=c(2,2,2,2),
+                col=c("black","red","black","coral3"), bg="white", cex=0.7)
```

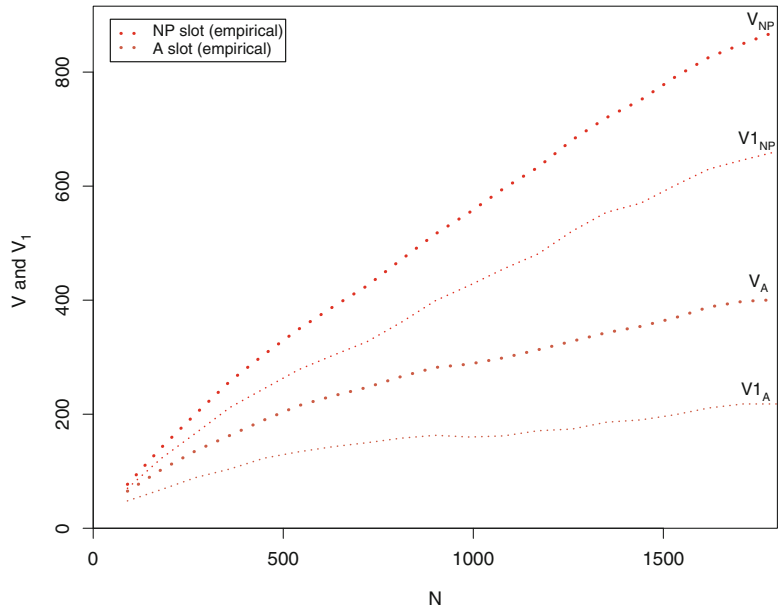


Fig. 9.6: Empirical VGCs of *A* and *NP* in *A as NP*

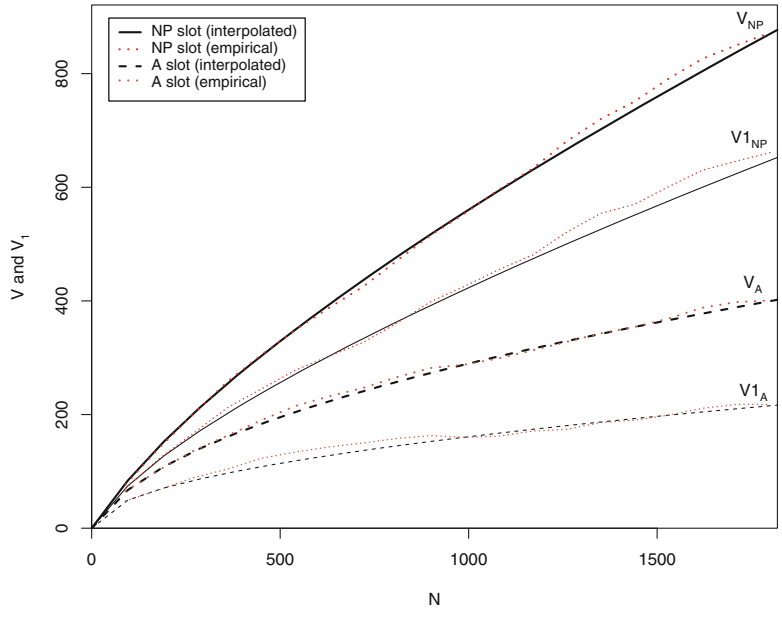


Fig. 9.7: Empirical and interpolated VGCs of *A* and *NP* in *A as NP*

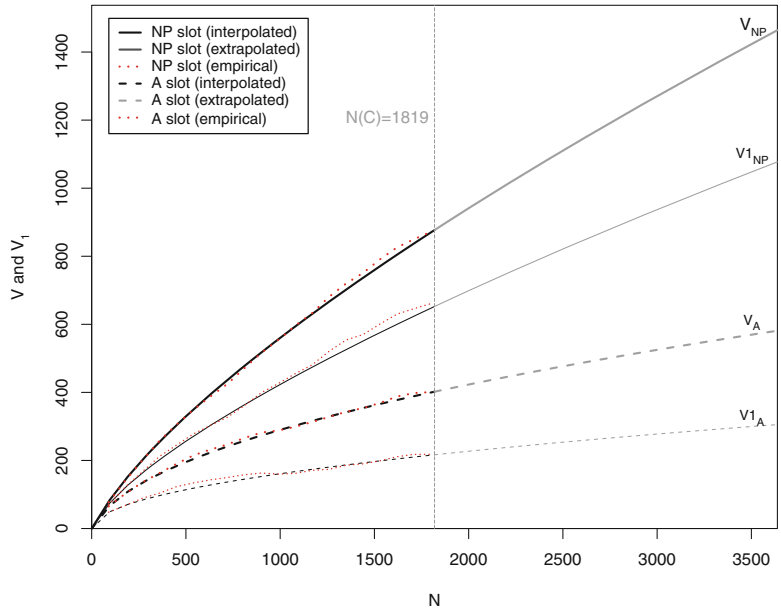


Fig. 9.8: Empirical, interpolated, and extrapolated VGCs of *A* and *NP* in *A as NP*

Finally, the code below plots Fig. 9.8. We include a vertical line to signal the limit of the "text", i.e. the starting point of the model's extrapolation.

```
> plot(df.int.fzm.np, df.ext.fzm.np, df.vgc.np, df.int.fzm.a, df.ext.fzm.a, df.vgc.a,
+      add.m=1, col=c("black", "grey60","red", "black", "grey60","red"),
+      lty=c(1,1,3,2,2,3), lwd=c(2,2,2,2,2,2),  main="", ylab="")
>
> ylab_name=expression(paste("V"," and"," V"[1],sep=""))
> title(ylab=ylab_name)
>
> abline=abline(v=1819, lty=5,lwd=0.5, col="grey60")
>
> text(1600, 1200, "N(C)=1819", cex=0.7, col="grey60")
> text(3500, 1470, expression("V"[NP]), cex=0.8)
> text(3500, 1100, expression("V1"[NP]), cex=0.8)
> text(3500, 610, expression("V"[A]), cex=0.8)
> text(3500, 330, expression("V1"[A]), cex=0.8)
>
> legend<-legend("topleft", inset=.025,c("NP slot (interpolated)","NP slot (extrapolated)",
+               "NP slot (empirical)", "A slot (interpolated)",
+               "A slot (extrapolated)","A slot (empirical)"),
+                lty=c(1,1,3,2,2,3), lwd=c(2,2,2,2,2,2),
+               col=c("black", "grey40","red","black", "grey60","red"),
+               bg="white", cex=0.7)
```

The VGCs confirm the hapax-based measure $\mathscr{P}$ (Sect. 9.4.1). This is far from surprising. One mathematical property of $\mathscr{P}$ is that it is the slope of the tangent to a VGC at its endpoint. It indicates the rate at which vocabulary increases at the end of a vocabulary growth curve. As expected the slope is steeper for NPs, which have more potential for variability, than for adjectives.

Exercise

9.1. Writing a function for association measures

Using what you have learned from Sect. 2.8.2, write a function named `assoc.chi()` to automatize the loop in Sect. 9.3.4. More advanced users should try to write an interactive function that prompts the user to:

- choose from two input file formats: `.rds` or `.txt`;
- enter the input file;
- input the frequency of the target word;
- input the corpus size in word tokens;
- select an output text file in which the results are saved;
- provide the path to the output file.

To write this interactive function, you will need to refer to the R documentation with regards to the following functions:

- `switch()`;
- `menu()`;
- `readline()`;
- `normalizePath()`.

References

Agresti, Alan. 2002. *Categorical Data Analysis*. Wiley Series in Probability and Statistics, 2nd ed. New York: Wiley-Interscience.

Allan, Lorraine G. 1980. A Note on Measurement of Contingency Between Two Binary Variables in Judgment Tasks. Bulletin of the Psychonomic Society 15 (3): 147–149.

Baayen, R. Harald. 2008. *Analyzing Linguistic Data: A Practical Introduction to Statistics Using R*. Cambridge: Cambridge University Press.

Baayen, Rolf Harald. 1989. *A Corpus-Based Approach to Morphological Productivity. Statistical Analysis and Psycholinguistic Interpretation*. Amsterdam: Centrum Wiskunde en Informatica.

Baayen, Rolf Harald. 1993. On Frequency Transparency and Productivity. In *Yearbook of Morphology 1992*, ed. Geert Booij, and Jaap van Marle, 181–208. Dordrecht, London: Kluwer.

Baayen, Rolf Harald. 2001. *Word Frequency Distributions*. Dordrecht: Kluwer Academic Publishers.

Baayen, Rolf Harald. 2009. Corpus Linguistics in Morphology: Morphological Productivity. In *Corpus Linguistics. An International Handbook*, ed. Anke Lüdeling and Merja Kytö, 899–919. Berlin: Mouton de Gruyter.

Baayen, Rolf Harald. 2011. Corpus Linguistics and Naive Discriminative Learning. *Brazilian Journal of Applied Linguistics* 11: 295–328.

Baayen, Rolf Harald, and Rochelle Lieber. 1991. Productivity and English Derivation: A Corpus-Based Study. *Linguistics* 29: 801–843.

Bauer, Laurie. 2001. *Morphological Productivity*. Cambridge: Cambridge University Press.

Bybee, Joan. 2010. *Language Usage and Cognition*. Cambridge: Cambridge University Press.

Bybee, Joan L. 2006. From Usage to Grammar: The Mind's Response to Repetition. *Language* 82 (4): 711–733.

Chomsky, Noam. 1957. *Syntactic Structures*. The Hague: Mouton.
Chomsky, Noam. 1962. *Aspects of the Theory of Syntax*. Cambridge, MA: MIT Press.
Church, Kenneth, and Patrick Hanks. 1990. Word Association Norms, Mutual Information, and Lexicography. *Computational Linguistics* 16 (1): 22–29.
Church, Kenneth, et al. 1991. Using Statistics in Lexical Analysis. In *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, ed. Uri Zernik, 115–164. Hillsdale: Lawrence Erlbaum.
Cressie, Noel, and Timothy R.C. Read. 1989. Pearson's χ^2 and the Loglikelihood Ratio Statistic G^2: A Comparative Review. *International Statistical Review* 57 (1): 19–43. ISSN: 03067734, 17515823. http://www.jstor.org/stable/1403582.
Davies, Mark. 2008–2012. *The Corpus of Contemporary American English: 450 Million Words, 1990–Present*. http://corpus.byu.edu/coca/.
Desagulier, Guillaume. 2014. Visualizing Distances in a Set of Near Synonyms: *rather*, *quite*, *fairly*, and *pretty*. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 145–178. Amsterdam: John Benjamins.
Desagulier, Guillaume. 2016. A Lesson from Associative Learning: Asymmetry and Productivity in Multiple-Slot Constructions. *Corpus Linguistics and Linguistic Theory* 12 (1): 173–219.
Dunning, Ted. 1993. Accurate Methods for the Statistics of Surprise and Coincidence. *Computational Linguistics* 19 (1): 61–74.
Ellis, Nick. 2006. Language Acquisition as Rational Contingency Learning. *Applied Linguistics* 27 (1): 1–24.
Ellis, Nick, and Fernando Ferreira Junior. 2009. Constructions and Their Acquisition: Islands and the Distinctiveness of Their Occupancy. *Annual Review of Cognitive Linguistics* 7: 187–220.
Evert, Stefan. 2005. The Statistics of Word Cooccurrences: Word Pairs and Collocations. PhD Dissertation. Universität Stuttgart. http://elibunistuttgartde/opus/volltexte/2005/2371/pdf/Evert2005phd.pdf (visited on 08/14/2015).
Evert, Stefan. 2009. Corpora and Collocations. In *Corpus Linguistics: An International Handbook*, ed. Anke Lüdeling and Merja Kytö, Vol. 2, 1212–1248. Berlin, New York: Mouton de Gruyter.
Evert, Stefan, and Marco Baroni. 2006. The `zipfR` Library: Words and Other Rare Events in R. Presentation at useR! 2006: The Second R User Conference, Vienna, Austria.
Evert, Stefan, and Marco Baroni. 2007. `zipfR`: Word Frequency Distributions in R. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics on Interactive Posters and Demonstration Sessions* (R Package Version 0.6–6 of 2012-04-03), Prague, Czech Republic, 29–32.
Firth, John. 1957. A Synopsis of Linguistic Theory, 1930–1955. In *Selected Papers of J.R. Firth 1952–1959*, ed. Frank Palmer, 168–205. London: Longman.
Gries, Stefan Thomas. 2013. 50-Something Years of Work on Collocations: What is or Should be Next...*International Journal of Corpus Linguistics* 18 (1): 137–166.
Gries, Stefan Thomas. 2014. *Coll. Analysis 3.5. A Script for R to Compute Perform Collostructional Analyses*. University of California, Santa Barbara. http://www.linguistics.ucsb.edu/faculty/stgries/teaching/groningen/coll.analysis.r (visited on 07/09/2016).
Gries, Stefan Thomas, and Anatol Stefanowitsch. 2004a. Co-Varying Collexemes in the *Into*-Causative. In *Language Culture and Mind*, ed. Michel Achard and Suzanne Kemmer, 225–236. Stanford: CSLI.
Gries, Stefan Thomas, and Anatol Stefanowitsch. (2004b). Extending Collostructional Analysis: A Corpus-Based Perspective on 'Alternations'. *International Journal of Corpus Linguistics* 9 (1): 97–129.
Hilpert, Martin. 2014. Collostructional Analysis: Measuring Associations Between Constructions and Lexical Elements. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 391–404. Amsterdam: John Benjamins.

Kennedy, Graeme. 2003. Amplifier Collocations in the British National Corpus: Implications for English Language Teaching. *TESOL Quarterly* 37 (3): 467–487.

Kilgarriff, Adam. 2001. Comparing Corpora. *International Journal of Corpus Linguistics* 6 (1): 97–133.

Kilgarriff, Adam. 2005. Language is Never Ever Ever Random. *Corpus Linguistics and Linguistic Theory* 1 (2): 263–276. http://dx.doi.org/10.1515/cllt.2005.1.2.263.

Kotze, Theunis J. van W., and D.V. Gokhale. 1980. A Comparison of the Pearson-χ^2 and Log-Likelihood-Ratio Statistics for Small Samples by Means of Probability Ordering. *Journal of Statistical Computation and Simulation* 12 (1): 1–13. doi:10 1080/00949658008810422. eprint: http://dx.doi.org/10.108/00949658008810422.

Lafon, Pierre. 1980. Sur la variabilité de la fréquence des formes dans un corpus *Mots* 1: 127–165.

Lafon, Pierre. 1981. Analyse lexicomètrique et recherche des cooccurrences. *Mots*: 95–148. ISSN: 0243-6450. http://www.persee.fr/web/revues/home/prescript/article/mots_0243-6450_1981_num_3_1_1041.

Lafon, Pierre. 1984. *Dépouillements et statistiques en lexicométrie*. Travaux de linguistique quantitative. Genève Paris: Slatkine, Champion.

Langacker, Ronald W. 1987. *Foundations of Cognitive Grammar: Theoretical Prerequisites*, Vol. 1. Stanford: Stanford University Press.

Langacker, Ronald W. 2008. *Cognitive Grammar : A Basic Introduction*. Oxford: Oxford University Press.

Lebart, Ludovic, André Salem, and Lisette Berry. 1998. *Exploring Textual Data*. Text, Speech and Language Technology. Dordrecht, Boston, London: Kluwer Academic Publishers.

Manning, Christopher D., and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge: MIT Press.

Muller, Charles. 1964. *Essai de statistique lexicale. L'Illusion Comique de Pierre Corneille*. Paris: Klincksieck.

Muller, Charles. 1973. *Initiation aux méthodes de la statistique linguistique*. Paris: Champion.

Muller, Charles. 1977. *Principes et méthodes de statistique lexicale*. Paris: Hachette.

Pavlov Ivan Petrovich. 1927. *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*. London: Oxford University Press/Humphrey Milford.

Pecina, Pavel. 2010. Lexical Association Measures and Collocation Extraction. *Language Resources and Evaluation* 44 (1): 137–158.

Pedersen, Ted. 1996. Fishing for Exactness. In *Proceedings of the South-Central SAS Users Group Conference*, 188–200. San Antonio, TX: SAS Users Group.

Plag, Ingo. 2003. *Word-Formation in English*. Cambridge: Cambridge University Press.

Read, Timothy R.C., and Noel A.C. Cressie. 1988. *Goodness-of-Fit Statistics for Discrete Multivariate Data*. Springer Series in Statistics. New York: Springer Verlag.

Rescorla, Robert A. 1968. Probability of Shock in the Presence and Absence of CS in Fear Conditioning. *Journal of Comparative and Physiological Psychology* 66: 1–5.

Schmid, Hans-Jörg. 2010. Does Frequency in Text Really Instantiate Entrenchment in the Cognitive System? In *Quantitative Methods in Cognitive Semantics: Corpus-Driven Approaches*, ed. Dylan Glynn and Kerstin Fischer, 101–133. Berlin, New York: Mouton de Gruyter.

Stefanowitsch, Anatol, and Stefan Thomas Gries. 2003. Collostructions: Investigating the Interaction of Words and Constructions. *International Journal of Corpus Linguistics* 8 (2): 209–243.

Stefanowitsch, Anatol, and Stefan Thomas Gries. 2005. Covarying Collexemes. *Corpus Linguistics and Linguistic Theory* 1 (1): 1–46.

Wagner, Allan R., Robert A. Rescorla. 1972. A Theory of Pavlovian Conditioning: Variations in the Effectiveness of Reinforcement and Nonreinforcement. In *Classical Conditioning II*, ed. Abraham H. Black, and William F. Prokasy, 64–99. New York: Appleton-Century-Crofts

Wiechmann, Daniel. 2008. On the Computation of Collostruction Strength: Testing Measures of Association as Expressions of Lexical Bias. *Corpus Linguistics and Linguistic Theory* 4 (2): 253.

Yates, Frank. 1984. Tests of Significance for 2×2 Contingency Tables. *Journal of the Royal Statistical Society. Series A (General)* 147 (3): 426–463.

Zeldes, Amir. 2012. *Productivity in Argument Selection: From Morphology to Syntax*. Berlin, New York: Mouton de Gruyter.

Zipf, George K. 1949. *Human Behavior and the Principle of Least Effort*. Cambridge: Addison-Wesley.

Chapter 10
Clustering Methods

Abstract In this chapter, I introduce clustering techniques Their aim is to form clusters of objects so that similar objects are grouped in the same clusters and different objects are grouped in different clusters. I also introduce the concept of a network graph which, although not a clustering technique, is a useful, related addition to your corpus linguistics tool repository.

10.1 Introduction

The inventory of clustering techniques is vast. This chapter covers five methods: principal component analysis, *t*-distributed Stochastic Neighbor Embedding, correspondence analysis, multiple correspondence analysis, and ascending hierarchical cluster analysis. A section on networks is appended to this chapter. Although not part of cluster analysis, graph theory (which networks are a product of) combines the assets of cluster plotting with information about how the group members are related.

Clustering is done thanks to unsupervised clustering techniques. They are used to help find structure in multivariate data thanks to the groupings of observations. They are exploratory (as opposed to predictive or explanatory) because the researcher makes no assumption as to what groupings are to be found. This is why they are said to be hypothesis-generating techniques.

10.1.1 Multidimensional Data

In Sect. 5.3, you learned how to compile a data frame from an annotated corpus. In the resulting data frame, each line corresponds to an observation. In statistical parlance, the observations are called "individuals" because the tables were originally based on surveys aimed at characterizing real people. They are also often referred to as "row variables". Each column is a variable describing the observations. Extra variables are generally added. The choice of these variables depends on what the linguist intends to show.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8_10

When several variables are summoned to characterize observations, and when you take into account the pairwise relationships between these variables, the resulting table is called "multivariate", "multifactorial", or "multidimensional".[1] You conduct a multifactorial study because you want to generalize over the profiles of (a) the observations, (b) the variables, or (c) both. Just about anything in language is influenced by several factors at the same time. As a result, multifactorial methods have become quite popular in linguistics.

10.1.2 Visualization

Multifactorial data sets are complex objects of which we want to get a synthetic view. We do it by finding clusters in the data. This is no trivial task because we need to make sure the clusters are valid. The reward is a graphic representation of the data set in terms of neat clusters.

In hierarchical cluster analysis, clusters are visualized in dimensionless dendrograms, the kinds of which pullulate in museums of natural history. Dendrograms, which are shaped like trees, can be read from bottom to top or from top to bottom. Fig. 10.1 shows a (famous) dendrogram.

Eigenvector-based methods output Euclidean maps that allow a graphic, pairwise interpretation of the dimensions of the multidimensional table. To understand the logic that underlies visualization in principal component analysis, correspondence analysis, and multiple correspondence analysis, consider Fig. 10.2. It provides four plane representations of the Eiffel Tower. These representations are two-dimensional projections of a three-dimensional object. Because the complexity of the represented object greatly surpasses any of its 2D projections, not all representations are equally satisfactory. Figs. 10.2a, b would not be adequate if you wanted to show the Eiffel Tower to someone who has never seen it before. Indeed, in these representations the distances are distorted: we have the impression that the highest observation deck at the center of Fig. 10.2b is close to the base of the each pillar when in fact they are separated by about 900 feet. In this respect, Fig. 10.2c is more satisfactory, but you have no indication as to the depth of the tower. Fig. 10.2d is probably the best 2D snapshot you could make of this 3D object. Although it is a partial projection, it captures the most important dimensions of the Eiffel Tower in an economical way.

The challenges that underlie the visualizations obtained with eigenvector-based methods are similar except that you do not seek to explore a 3D object in real space, but a cloud of points from a data set in the form of a *rows* $\times$ *columns* table with as many dimensions as there are columns. Even though the dimensions of a data table are eventually projected in a Euclidean space, they should not be mixed up with spatial dimensions. A table with K variables has K dimensions. The cloud of data points is positioned in a space $\mathbb{R}$ of K dimensions: $\mathbb{R}^K$. To allow for easier interpretation, eigenvector-based methods decompose the cloud into orthogonal planes.

[1] Experience tells me that a unified definition for each term is hard to find. I have come to use the term "multidimensional" to describe a data set, and "multifactorial" to describe a statistical method that handles multidimensional data. Not every statistician will agree, but many will.

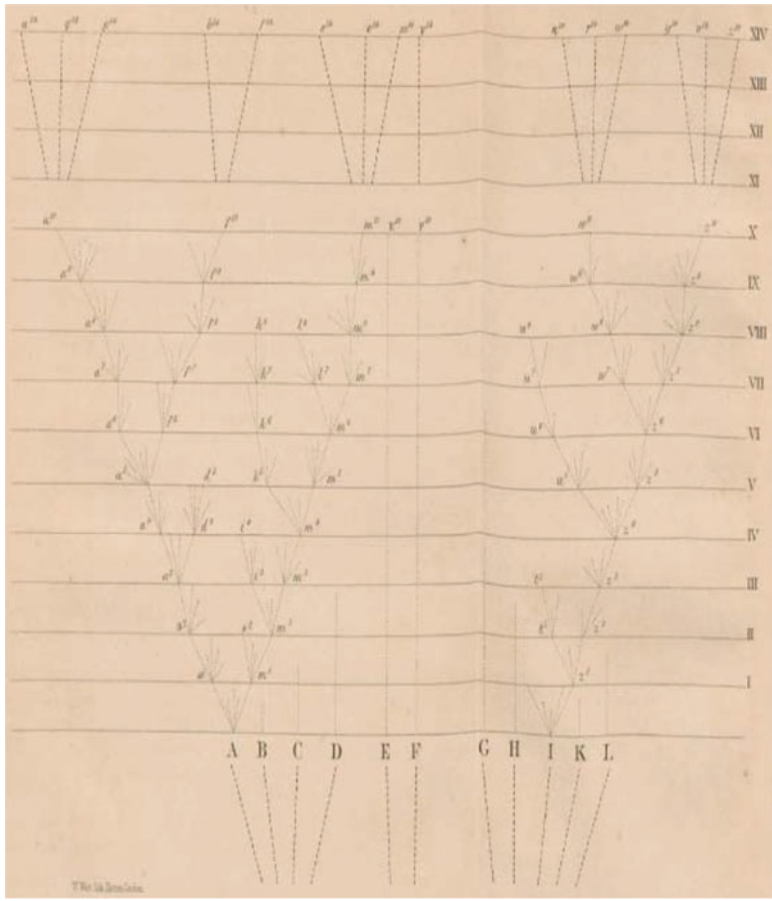


Fig. 10.1: A phylogenetic tree by Darwin (1859, p. 117)

A successful projection selects a viewpoint that manages to represents the cloud of data points with K dimensions on a low-dimensional plane while remaining as faithful as possible to $\mathbb{R}^K$. This is achieved by rotating the cloud of data points axially so as to minimize the distortion of distances between the individuals, just like you would rotate a 3D model of the Eiffel Tower so as to achieve a satisfactory 2D snapshot.

However, the kinds of data sets that we use in multifactorial linguistics are generally far more complex. In this respect, even though eigenvalue-based methods try to reduce the gap between the actual and projected distances between points, the visualizations that you obtain are often closer to a night sky map than the Eiffel Tower. Two stars may appear close on a 2D plane but one of them may actually be several lightyears behind the other in the universe. This means that you will often need more than one planes to describe your data accurately.

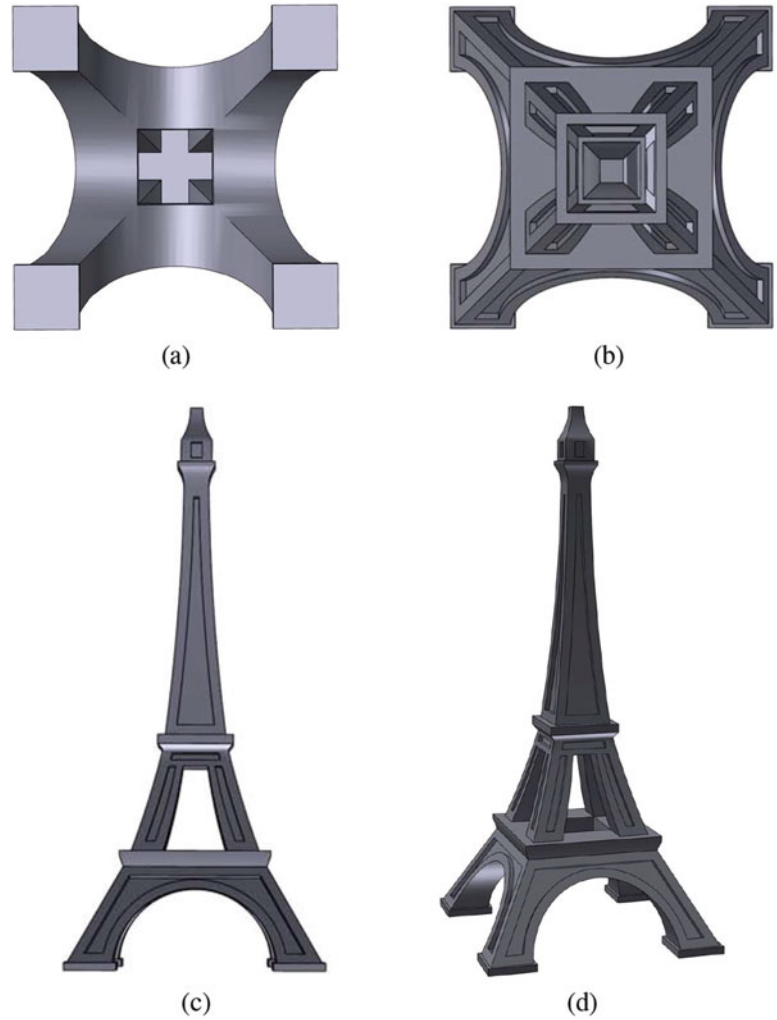


Fig. 10.2: The Eiffel Tower viewed from different angles. (**a**) A view from below; (**b**) a view from above; (**c**) a front view; (**d**) a three-quarter view

10.2 Principal Component Analysis

Principal component analysis (henceforth PCA) appeared in the early twentieth century (Pearson 1901) and has since acquired a name and fame in the field of data analysis. For this reason, PCA is given pride of place in most publications on clustering by being introduced before the other methods that it conceptually inspired, such as correspondence analysis and multiple correspondence analysis (see Sects. 10.4 and 10.5 below). I will make no exception here. However, even though the principles of PCA are simple to understand, its mathematical foundations are, I find, more demanding than those of (multiple) correspondence analysis. I

will therefore not dwell upon these foundations and focus instead on what is important to know to interpret the output of PCA.[2]

10.2.1 Principles of Principal Component Analysis

PCA it takes as input a data table T of I individuals or observations (rows) and K variables (columns). This table consists of ratio-scale data (such as means, reaction times, formant frequencies, etc.) and may also contain categorical/nominal data. The method consists in reducing the dimensionality of the table by decomposing the total variance of the table into a limited number of components.[3] More precisely, PCA extracts the important information from the table and expresses this information in the form of a handful of new orthogonal variables called principal components.

There are three reasons why you want to run a PCA on a data table: (a) you want to compare the profiles of individuals so that similar individuals cluster together, (b) you want to compare the profiles of the variables, so that correlated variables cluster together, and/or (c) you want to explore the link between the individuals and the variables.

There are several packages and functions dedicated to PCA in R: e.g. `princomp()` and `prcomp()` from the `stats` package, `ggbiplot()` from the `ggbiplot` package (which is itself based on `ggplot2`), `dudi.pca()` from the `ade4` package, and `PCA()` from the `FactoMineR` package (Husson et al. 2015).[4]

In this chapter, I will use `FactoMineR` because its functions and arguments are simple to understand, and because it is widely documented and supported. Also, functions from this package will be used to perform PCA, correspondence analysis, and multiple correspondence analysis. You should therefore install `FactoMineR` now.

```
> install.packages("FactoMineR")
```

10.2.2 A Case Study: Characterizing Genres with Prosody in Spoken French

What follows is based on Lacheret et al. (to appear a). The data set presented here was compiled from Rhapsodie, a corpus of spoken French annotated for syntax and prosody (Lacheret et al. to appear b). The motivation for building the data set is to characterize spoken samples in terms of prosodic features and stylistic categories.

The data set is available in your `chap10` subfolder.

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap10/spoken_french.rds") # Windows
> data <- readRDS("/CLSR/chap10/spoken_french.rds") # Mac
```

[2] Readers who want to know more about the mathematical underpinnings of PCA should read Baayen (2008, section 5.1.1), Abdi and Williams (2010), and Husson et al. (2011, chapter 1).

[3] The variance is the mean square deviation around the average value. It measures the dispersion of values around the mean.

[4] Be careful, these functions are not identical. Some of them perform PCA based on "loadings" whereas others perform PCA based on the inspection of correlations between the variables and the principal components. The latter are more flexible because they allow for the introduction of supplementary variables. This will become clearer in a few paragraphs.

It consists of 57 observations and 12 variables. The variables break down into six quantitative variables and six qualitative (nominal) variables. The quantitative variables are the following:

- `pauses`: the mean number of pauses per second;
- `overlaps`: the mean number of conversational overlaps per second;
- `euh`: the mean number of gap fillers per second ("euh" is the French equivalent of "er" or "um", when the speaker is not sure what to say);
- `prom`: the mean number of prosodic prominences per second;
- `PI`: the mean number of periods per second;
- `PA`: the mean number of intonation packages per second.

The qualitative variables provide information as to the following elements:

- `genre`: what genre the speech sample represents;
- `interactivity`: how interactive the speech sample is;
- `social_context`: whether the speech sample public or private;
- `event_structure`: whether the speech sample is part of a dialogue or a monologue;
- `channel`: whether the speech sample is part of a broadcast or a face-to-face conversation;
- `planning_type`: whether the speech sample is planned, semi-spontaneous, or spontaneous.

The above information is available when you examine the structure of `data`.

```
> str(data)
'data.frame': 57 obs. of  12 variables:
 $ pauses         : num  0.264 0.42 0.354 0.277 0.286 ...
 $ overlaps       : num  0.1242 0.1066 0.0981 0.1127 0.0664 ...
 $ euh            : num  0.1353 0.1032 0.0323 0.119 0.2336 ...
 $ prom           : num  1.79 1.8 1.93 2.29 1.91 ...
 $ PI             : num  0.278 0.326 0.335 0.303 0.219 ...
 $ PA             : num  1.54 1.75 1.76 1.79 1.69 ...
 $ subgenre       : Factor w/ 5 levels "argumentation",..: 1 1 2 2 2 1 5 5 2 5 ...
 $ interactivity  : Factor w/ 3 levels "interactive",..: 1 1 3 1 3 1 2 2 1 2 ...
 $ social_context : Factor w/ 2 levels "private","public": 1 1 1 1 1 1 1 1 1 1 ...
 $ event_structure: Factor w/ 2 levels "dialogue","monologue": 1 1 1 1 1 1 1 1 1 1 ...
 $ channel        : Factor w/ 2 levels "broadcasting",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ planning_type  : Factor w/ 3 levels "planned","semi-spontaneous",..: 2 2 3 2 3 2 3 3 3 3 ...
```

The motivation for this data frame is twofold. We want to compare (a) the profiles of the corpus samples based on the variables, and (b) the variables.

The shape of the cloud of points will be based on the quantitative variables. At this stage, we shall ignore the qualitative variables so as to focus on the construction of the cloud. We shall return to them to describe the shape of the cloud.

To understand why PCA is useful, let us examine the pairwise relationships between the quantitative variables. With `cor()`, you obtain a correlation matrix (the correlation coefficients are rounded to two decimal places with `round()`).

```
> round(cor(data[,1:6]), 2)
         pauses overlaps   euh  prom    PI    PA
pauses     1.00    -0.07 -0.18  0.05  0.56  0.57
overlaps  -0.07     1.00 -0.16  0.11  0.23  0.25
euh       -0.18    -0.16  1.00 -0.26 -0.46 -0.24
prom       0.05     0.11 -0.26  1.00  0.13  0.60
PI         0.56     0.23 -0.46  0.13  1.00  0.44
PA         0.57     0.25 -0.24  0.60  0.44  1.00
```

A positive figure indicates a positive correlation, whereas a negative figure indicates a negative correlation. All the measures are correlated, but not all correlations are significant. In some cases, the correlation is statistically significant ($p < 0.05$),

```
> cor.test(data$PA, data$prom)

Pearson's product-moment correlation

data:  data$PA and data$prom
t = 5.5774, df = 55, p-value = 7.666e-07
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.4037142 0.7449510
sample estimates:
      cor
0.6010541
```

whereas in some other cases it is not ($p > 0.05$).

```
> cor.test(data$pauses, data$euh)

Pearson's product-moment correlation

data:  data$pauses and data$euh
t = -1.3234, df = 55, p-value = 0.1912
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.41714324  0.08897032
sample estimates:
       cor
-0.1756699
```

To visualize the pairwise correlations between variables, use the function `pairs()` (Fig. 10.3).

```
> pairs(data[,1:6], cex=0.6)
```

In Fig. 10.3, each scatterplot corresponds to the intersection of two variables. The labels are given in the diagonal of the plot. Because there are six variables, and because the row and column vectors are symmetric, there are $\frac{6\times5}{2} = 15$ pairwise relationships to inspect, which is cumbersome.

10.2.3 How PCA Works

PCA offers an alternative by reducing the dimensionality of the data table so that you do not need to review each pairwise correlation. It does so by computing the total variance of the table and decomposing this variance into a few meaningful components. The principal components are linear combinations of the original variables turned into new variables.

Before you run `PCA()`, you should consider standardizing (i.e. centering and scaling) the variables. If your table contains measurement in different units (for example hertz, seconds, and percentages), standardizing is compulsory. By default, `PCA()` standardizes the variables. In our table, all the measurements are in the same unit (relative frequency per second). Although we could dispense with standardizing the variables, I recommend that you do. This prevents variables with a high standard deviation from having too strong an influence relative to the variables with a lower standard deviation. Thus, the same weight is given to all the quantitative variables. If you choose not to standardize your variables, include the argument `scale.unit=FALSE` in your `PCA()` call.

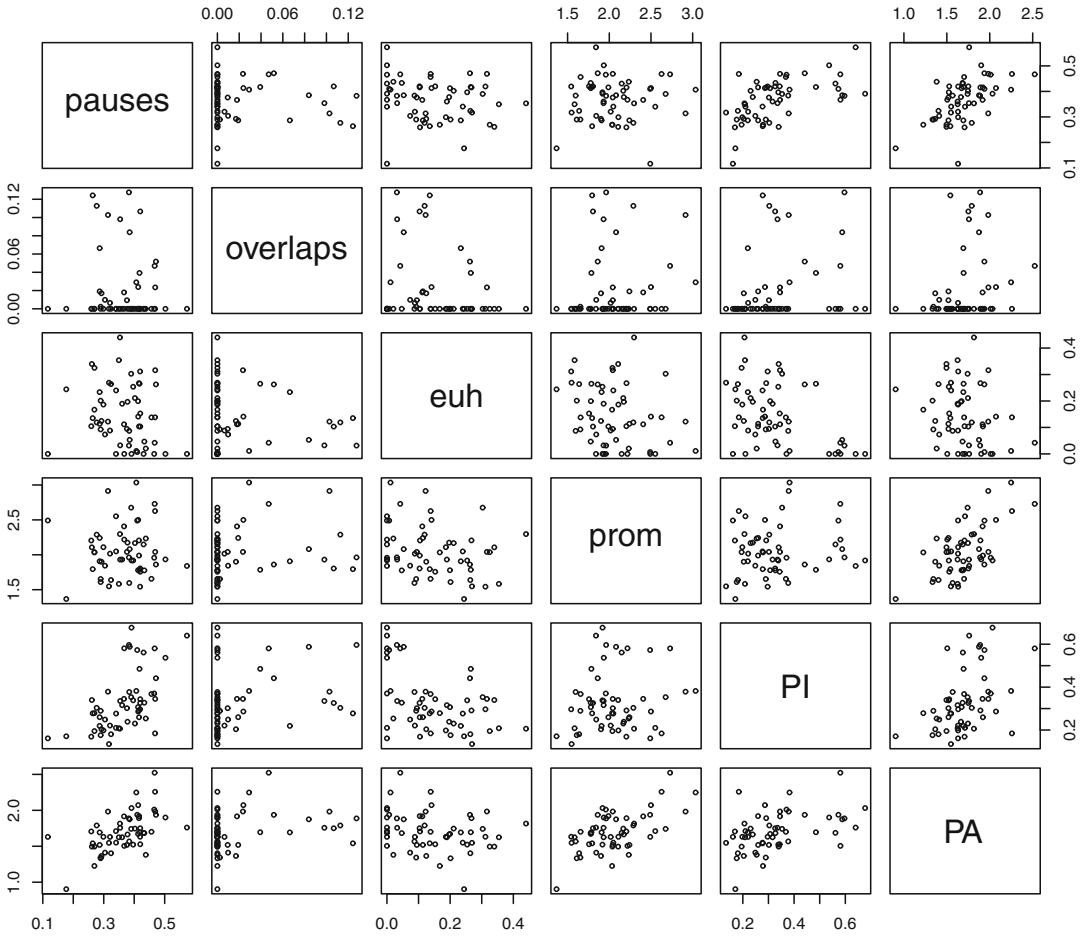


Fig. 10.3: Pairwise correlations between quantitative variables

By default, the `PCA()` function produces two graphs based on the first two components: the graph of variables and the graph of individuals. For the time being, I recommend that you block these plots by specifying `graph=FALSE`. We will plot each graph individually later, once we know how to interpret them.

```
> library(FactoMineR)
> pca.object <- PCA(data[,1:6], graph=FALSE) # remove graph=FALSE to display the default plots
> pca.object # inspect
**Results for the Principal Component Analysis (PCA)**
The analysis was performed on 57 individuals, described by 6 variables
*The results are available in the following objects:

   name               description
1  "$eig"             "eigenvalues"
2  "$var"             "results for the variables"
3  "$var$coord"       "coord. for the variables"
4  "$var$cor"         "correlations variables - dimensions"
5  "$var$cos2"        "cos2 for the variables"
6  "$var$contrib"     "contributions of the variables"
```

```
7  "$ind"             "results for the individuals"
8  "$ind$coord"       "coord. for the individuals"
9  "$ind$cos2"        "cos2 for the individuals"
10 "$ind$contrib"     "contributions of the individuals"
11 "$call"            "summary statistics"
12 "$call$centre"     "mean of the variables"
13 "$call$ecart.type" "standard error of the variables"
14 "$call$row.w"      "weights for the individuals"
15 "$call$col.w"      "weights for the variables"
```

The object contains several elements, the first of which (`eig`) is worth inspecting because it helps you determine which components are important. The importance of a component corresponds to the proportion of the total variance of the table that it explains (this total variance is called inertia).

```
> round(pca.object$eig, 2)
       eigenvalue percentage of variance cumulative percentage of variance
comp 1       2.51                 41.90                             41.90
comp 2       1.15                 19.25                             61.15
comp 3       1.01                 16.88                             78.03
comp 4       0.83                 13.77                             91.79
comp 5       0.33                  5.56                             97.35
comp 6       0.16                  2.65                            100.00
```

The first principal component explains the largest share of the total variance of the data table. The second component is orthogonal to the first component. It explains the second largest share of inertia. The remaining components are computed likewise.

Next, we plot the proportions of inertia (also known as eigenvalues) explained by the principal components (Fig. 10.4).

```
> barplot(pca.object$eig[,1], names.arg=paste("comp ",1:nrow(pca.object$eig))) # barplot
```

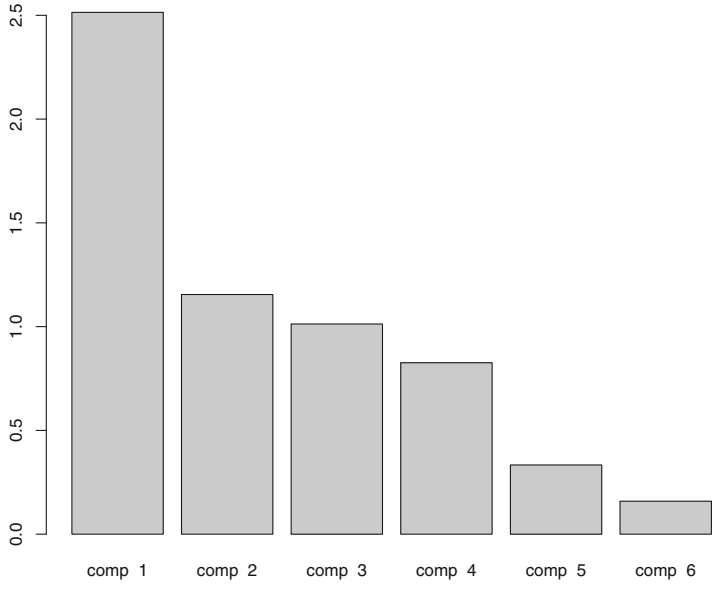


Fig. 10.4: A barplot showing the eigenvalues associated with each component

To select the most important components, we look for discontinuities in the plot as we move from left to right. In Fig. 10.4, the first major discontinuity occurs after the first component. This component alone accounts for 41.9% of the variance which, given the ratio between the number of observations and the number of columns, expresses a significant structure of the data in PCA. A second discontinuity occurs between the fifth and the fourth components. The first four components account for almost all the variance (91.8%). In other words, PCA has reduced the number of dimensions to inspect from 6 to 4.

It is now time to visualize the PCA output with `plot()` and its fine-tuning options. A combination of the `shadow` and `autoLab` arguments guarantees that the labels will be legible if there are many data points. By default, the first two principal components are plotted. They correspond to the most optimal plane because the distances between individuals are the least distorted and because the variance of the projected data is maximized.

```
> pca.object <- PCA(data[,1:6], graph=FALSE) # no plot is produced
> plot(pca.object,shadow=T, autoLab="yes", choix="var",new.plot=F, title="") # variables
> plot(pca.object, shadow=T, autoLab="yes", title="") # individuals
```

The first graph shows the variables in $\mathbb{R}^K$ (Fig. 10.5a), whereas the second one shows the cloud of individuals in $\mathbb{R}^I$ (Fig. 10.5b). The first graph is a key to interpreting the second graph.

In PCA, correlations are interpreted geometrically. Fig. 10.5a shows a correlation circle. Each active variable is plotted inside a circle of radius 1 using correlation coefficients as coordinates. These coordinates are available from `pca.object`.[5]

```
> pca.object$var$coord[,1:2]
              Dim.1       Dim.2
pauses    0.6767341 -0.62190814
overlaps  0.3292927  0.55179602
euh      -0.5789986 -0.09553468
prom      0.5493958  0.58167708
PI        0.7718681 -0.31577079
PA        0.8455460  0.12774240
```

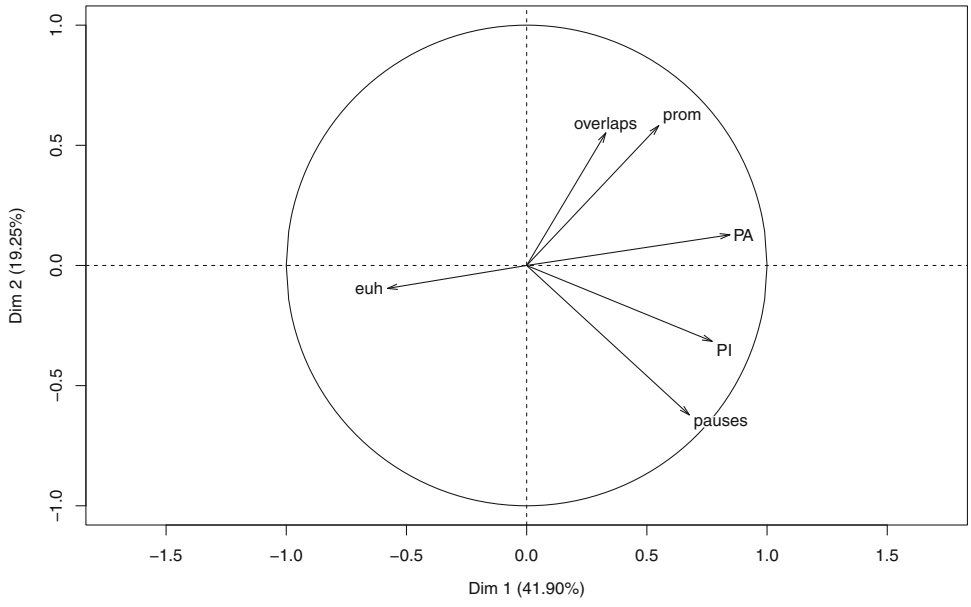
The variables appear as a solid arrows.[6] We can think of each arrow as pulling away from the intersection of the horizontal and vertical axes. The longer the arrow, the harder it pulls. With all the arrows pulling together, PCA considers their overall effect.

In the first component of variability, there is a contrast between the variable `euh`, which points to the left, and the other variables, which point to the right. In the second component, there is a contrast between the variables `overlaps`, `prom`, and `PA` (top) and `euh`, `PI`, and `pauses` (bottom). When we combine these first two components, we expect the corpus samples characterized by more overlaps, prominences, or intonation packages to cluster in the upper right part of the plane. The corpus samples characterized by more periods or pauses are expected to cluster in the lower right part of the plane. The corpus samples characterized by more gap fillers are expected to cluster in the lower left part of the plane. The corpus samples expected to be found in the upper left corner of the plane are characterized by smaller scores with respect to the other variables, the detail of which we shall come back to later. The actual projection of these corpus samples is shown in Fig. 10.5b.

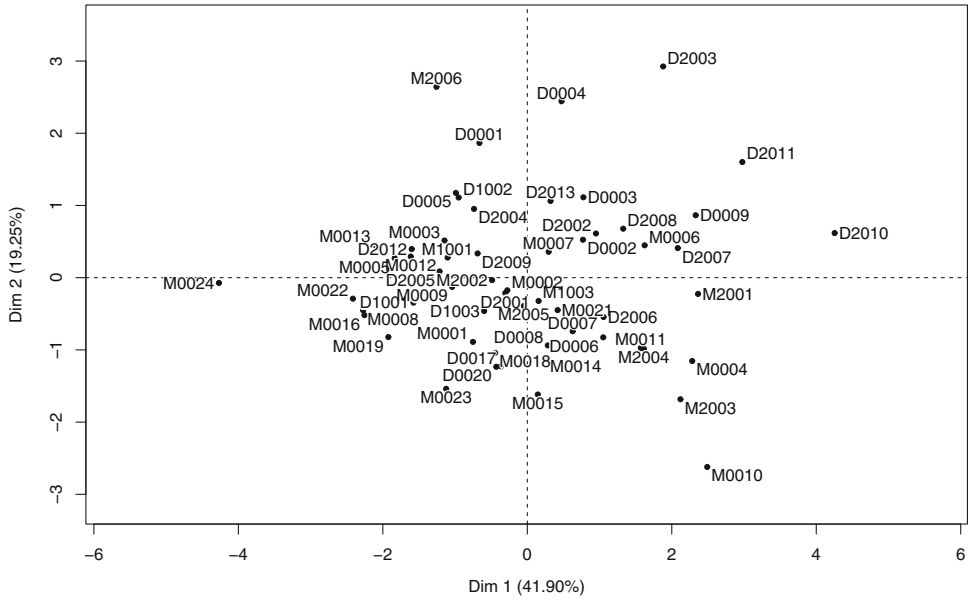
Unless we know exactly what the corpus samples consist of, it is hard to generalize from Fig. 10.5b. This is what supplementary variables are for. Let us run `PCA()` again. This time, we include the qualitative variables and declare them as supplementary.

[5] Because our variables are standardized, the coordinates correspond to the coefficients of the correlations between the variables and the components.

[6] If there were supplementary variables (see below), they would appear as dashed arrows.



(a)



(b)

Fig. 10.5: PCA biplots (components 1 and 2). (**a**) Variables; (**b**) individuals

```
> pca.object.2 <- PCA(data, quali.sup=c(7:12), graph=FALSE)
```

The insights offered by the inclusion and projection of supplementary information is what makes PCA and related methods attractive. Supplementary rows and/or columns are projected in the plane obtained from the PCA performed on the original data table. Supplementary data can be continuous or categorical (our supplementary variables are categorical). Bear in mind that supplementary elements do not contribute to the construction of the components. They are projected only after inertia has been computed from the active elements. This does not affect the construction of the components.

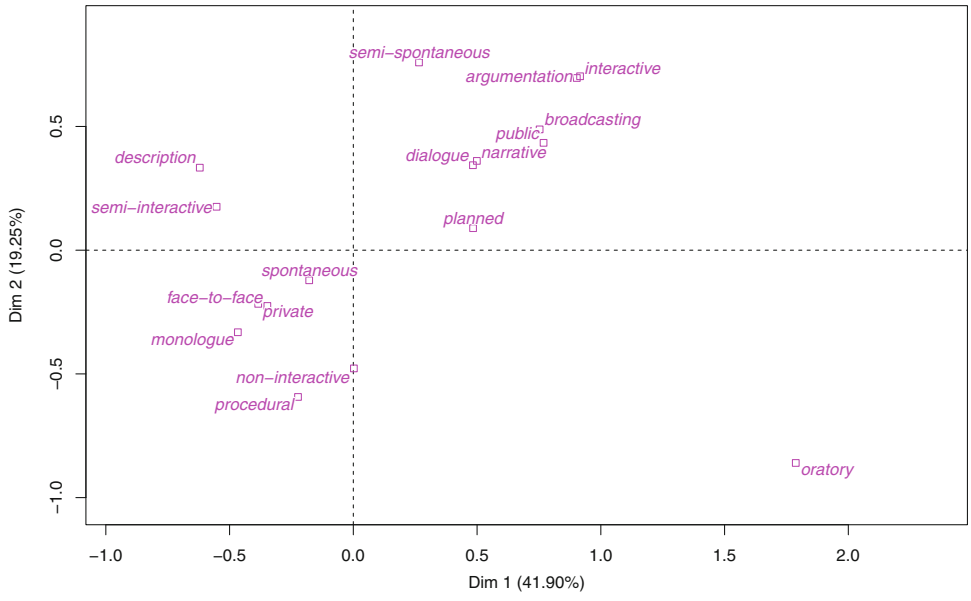
```
> round(pca.object.2$eig, 2)
       eigenvalue percentage of variance cumulative percentage of variance
comp 1       2.51                 41.90                            41.90
comp 2       1.15                 19.25                            61.15
comp 3       1.01                 16.88                            78.03
comp 4       0.83                 13.77                            91.79
comp 5       0.33                  5.56                            97.35
comp 6       0.16                  2.65                           100.00
```

Rather than plot the individuals, you may now focus on the categories of the supplementary variables.

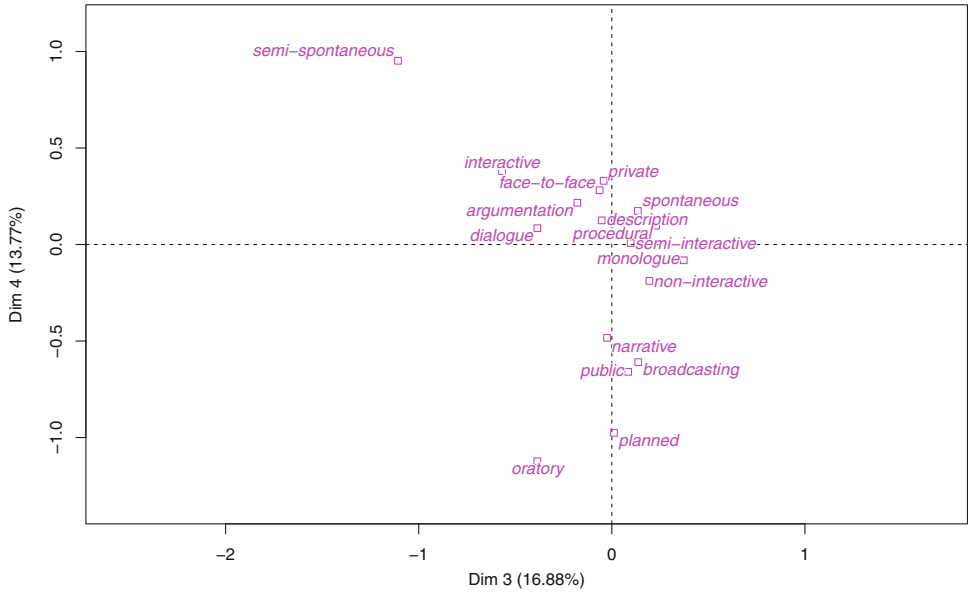
Neat clusters appear in Fig. 10.6a. As opposed to descriptive, semi-interactive samples, oratory samples are characterized by a high number of intonation periods and very frequent pauses. There is a neat contrast between the samples in the upper-right corner and those in the lower-left corner.

Inspection of Fig. 10.6b reveals that the atypical profile of `oratory` samples should be toned down with respect to the third and fourth components. In contrast, the profile of `semi-spontaneous` stands out with respect to these components.

```
> dimdesc(pca.object.2)
$Dim.1
$Dim.1$quanti
          correlation      p.value
PA          0.8455460 1.288448e-16
PI          0.7718681 2.085404e-12
pauses      0.6767341 7.523535e-09
prom        0.5493958 9.606625e-06
overlaps    0.3292927 1.237681e-02
euh        -0.5789986 2.378971e-06

$Dim.1$quali
                        R2     p.value
social_context  0.11750449 0.009047008
subgenre        0.21039303 0.013918880
channel         0.10402383 0.014410761
event_structure 0.08967506 0.023637198
interactivity   0.11287505 0.039408460

$Dim.1$category
                Estimate     p.value
public         0.5765062 0.009047008
broadcasting   0.5501019 0.014410761
oratory        1.3184753 0.019048756
dialogue       0.4749011 0.023637198
interactive    0.7941458 0.024182741
description   -1.0895818 0.023950148
monologue     -0.4749011 0.023637198
face-to-face  -0.5501019 0.014410761
private       -0.5765062 0.009047008


$Dim.2
$Dim.2$quanti
          correlation      p.value
```



(a)



(b)

Fig. 10.6: PCA biplots (supplementary variables in $\mathbb{R}^J$). (**a**) Components 1 and 2; (**b**) components 3 and 4

```
prom       0.5816771 2.082481e-06
overlaps   0.5517960 8.621097e-06
PI        -0.3157708 1.672159e-02
pauses    -0.6219081 2.422954e-07

$Dim.2$quali
                      R2     p.value
subgenre        0.25614407 0.003490792
interactivity   0.18617948 0.003839487
event_structure 0.09864320 0.017348845
channel         0.09529872 0.019469480
social_context  0.08151312 0.031335204

$Dim.2$category
                  Estimate     p.value
interactive      0.5694425 0.010168067
dialogue         0.3375553 0.017348845
broadcasting     0.3568325 0.019469480
public           0.3254128 0.031335204
argumentation    0.7085504 0.034595162
private         -0.3254128 0.031335204
face-to-face    -0.3568325 0.019469480
monologue       -0.3375553 0.017348845
procedural      -0.5806223 0.002684967
non-interactive -0.6113854 0.002492045


$Dim.3
$Dim.3$quanti
         correlation      p.value
prom       0.5122576 4.633616e-05
euh        0.3952687 2.341783e-03
PA         0.3492439 7.750895e-03
PI        -0.3472536 8.132105e-03
overlaps  -0.5709713 3.520492e-06

$Dim.3$quali
                      R2     p.value
event_structure 0.1418851 0.003876516
planning_type   0.1188896 0.032795879

$Dim.3$category
                   Estimate     p.value
monologue         0.3791308 0.003876516
interactive      -0.4764462 0.027693050
semi-spontaneous -0.7871007 0.009385061
dialogue         -0.3791308 0.003876516
```

10.3 An Alternative to PCA: t-SNE

We live in the age of big data. Big data implies very large data frames, and individuals by the thousands. In this case, plotting the graph individuals with PCA may be cumbersome as you might expect data points to clutter the graph. Another problem is when the number of descriptive variables is also very large. In this section, I illustrate this problem and how to fix it with a case study involving large data frames of annotated corpus data.

Whether manual or (semi-)automatic, the semantic annotation of a comprehensive corpus or a large dataset is a hard and time-consuming task. One long-standing challenge is the resolution of lexical and syntactic ambiguities. One option favored by corpus semanticists is to use a set of predetermined classes, such as Levin (1993) for verbs or Dixon and Aikhenvald (2004) for adjectives.

Because such semantic classes are generally broad and determined a priori, they hardly ever match the ad hoc contextual meanings of their targets. By way of example, Tab. 10.1 is a sample dataset compiled from the British National Corpus (2007). It contains a sample of adjectives that occur in *quite/rather* constructions. Each adjective is automatically annotated with the UCREL Semantic Analysis System (USAS). As expected, the contextual meanings of highly polysemous adjectives like *hot* and *cold* are poorly captured by the limited range of possibilities of the generic tagset: e.g. *hot* in *a rather hot seller* has little to do with 'Temperature'.

Table 10.1: A sample data frame with USAS annotation of adjectives (BNC XML)

corpus file	context	adjective	semantic tag	semantic class
K1J.xml	*quite a hot shot*	hot	O4.6+	Temperature_Hot_on_fire
G2W.xml	*a rather hot seller*	hot	O4.6+	Temperature_Hot_on_fire
KRT.xml	*a quite clear position*	clear	A7+	Likely
J0V.xml	*quite a clear understanding*	clear	A7+	Likely
CHE.xml	*quite a clear view*	clear	A7+	Likely
FEV.xml	*quite a clear picture*	clear	A7+	Likely
EWR.xml	*a quite clear line*	clear	A7+	Likely
CRK.xml	*quite a clear stand*	clear	A7+	Likely
HA7.xml	*a rather clouded issue*	clouded	O4.3	Colour_and_colour_patterns
KPV.xml	*quite a cold day*	cold	O4.6-	Temperature_Cold
G3B.xml	*a rather cold morning*	cold	B2-	Disease
AB5.xml	*a rather cold person*	cold	O4.6-	Temperature_Cold
CDB.xml	*rather a cold note*	cold	O4.6-	Temperature_Cold
K23.xml	*a rather colder winter*	colder	O4.6–	Temperature_Cold

Two cutting-edge unsupervised learning algorithms have offered a solution to this problem: `word2vec` (Mikolov et al. 2013a,b,c) and `GloVe` (Pennington et al. 2014). Based on neural networks, they *(a)* learn word embeddings that capture the semantics of words by incorporating both local and global corpus context, and *(b)* account for homonymy and polysemy by learning multiple embeddings per word. Once trained on a very large corpus, these algorithms produce distributed representations for words in the form of vectors. Words or phrases from the vocabulary are mapped to vectors of real numbers (Tab. 10.2). Words with similar vector representations have similar meanings.

The data set `adjectives.vectors.rds` contains 1108 adjectives (rows) and each is characterized by a 300-dimensional word vector representation (columns).

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap10/adjectives.vectors.rds") # Windows
> data <- readRDS("/CLSR/chap10/adjectives.vectors.rds") # Mac
```

Because the data frame is very wide (300 columns), we display only the first ten rows and the first nine columns.

```
> head(data[1:10,1:9])
                V1       V2        V3        V4        V5        V6       V7       V8        V9
able       -0.274940  0.23719  0.058772 -0.043835 -0.018766  0.155180 -4.33010  0.21271  0.174290
abnormal   -0.211740 -0.27585 -0.929810  0.279370 -0.462320 -0.177400 -2.24480  1.57590  0.026833
abrupt     -0.100270 -0.24864  0.046093  0.224550 -0.827240 -0.150440 -1.74370  1.02900  0.132330
abstract    0.079376 -0.78241 -0.103870  0.625730  0.034500 -0.029305 -1.72160  0.43605  0.189480
```

Table 10.2: Snapshot of a word-vector matrix of some adjectives from the BNC

adjectives	V1	V2	V3	V4	V5	V6	...
gloomy	−0.36405	−0.44487	−0.33327	−0.16695	−0.52404	0.31066	...
sacred	0.60337	−0.20526	−0.042822	−0.33008	−0.68957	0.26654	...
jaundiced	−0.32168	−0.58319	−0.34614	−0.12474	0.10368	0.1733	...
loud	0.24615	−0.24904	−0.18212	−0.14834	−0.06532	−0.3393	...
memorable	0.30206	0.20307	0.062304	0.66816	0.048326	0.034361	...
justified	−0.080959	−0.23694	−0.43372	−0.31442	−0.31528	0.0057226	...
scant	−0.14467	−0.29329	0.10832	−0.11123	−0.57925	−0.27022	...
continuous	−0.15253	−0.082764	−0.40871	−0.53719	0.0822	−0.31482	...
imposing	0.32043	0.155	−0.10547	−0.23157	−0.35657	−0.097553	...
weighty	0.085281	0.015087	0.58454	0.0094917	−0.082617	0.36811	...
...	...	...	...	...	...	...	...

```
abstracted  0.230450 -0.31227 -0.305120 -0.186680  0.036875  0.124390 -0.69434  0.03134  0.540250
absurd     -0.588760 -0.50195 -0.148390  0.271660 -0.283130  0.313740 -2.13350 -0.10671 -0.374170
               V10
able         0.09212
abnormal     0.25169
abrupt      -0.24750
abstract     0.28779
abstracted   0.13886
absurd      -0.59916
```

On the one hand, word vectors capture context-sensitive information. On the other hand, the number of dimensions is so high that we would have to inspect many dimensions in a pairwise fashion if we inspected a graph based on a PCA of the data frame.

Laurens van der Maaten has devised a dimensionality reduction technique known as t-Distributed Stochastic Neighbor Embedding (t-SNE) (van der Maaten and Hinton 2008). He claims that t-SNE dimensionality reduction is particularly well suited for the visualization of high-dimensional datasets. The t-SNE algorithm is implemented in R in two packages: `tsne` and `Rtsne`.[7] I use the latter, with the dedicated function `Rtsne()`.

```
> install.packages("Rtsne") # install the Rtsne package
> library(Rtsne) # load the package
```

The argument `initial_dims` specifies the number of word-vector columns.

```
> rtsne <- Rtsne(as.matrix(data), initial_dims=300)
> str(rtsne)
List of 7
 $ theta     : num 0.5
 $ perplexity: num 30
 $ N         : int 1108
 $ origD     : int 300
 $ Y         : num [1:1108, 1:2] 0.423 -5.541 3.93 -2.07 -1.544 ...
 $ costs     : num [1:1108] 0.00419 0.00146 0.00171 0.00143 0.00161 ...
 $ itercosts : num [1:20] 70 69.5 69.4 70 70 ...
```

[7] `Rtsne` is a wrapper for van der Maaten's implementation of t-SNE via Barnes-Hut approximations.

`Rtsne()` outputs a list whose `Y` component we want to plot. First, call the plot with `plot()` and indicate `type="n"` so that no data point is plotted. Instead, use the row names to label each point with `text()` and its `labels` argument. Because there are many adjectives to plot, reduce the font size with the `cex` argument.

```
> plot(rtsne$Y, type="n", main="Barnes-Hut t-SNE")
> text(rtsne$Y, labels=rownames(data), cex=0.5)
```

You may find the plot in Fig. 10.7 too daunting because of the accumulation of words scattered across the euclidean space. If so, you may want to partition the data points into classes with a method known as *k*-means clustering. *K*-means partitions the observations into *k* classes. Each observation is part of the cluster with the nearest mean. We select the desired number of clusters via the `centers` argument. Here, we choose to have ten clusters. In base R, you perform *k*-means clustering with the `kmeans()` function, which outputs a list.

```
> km.clustering <- kmeans(data, centers=10)
> str(km.clustering)
List of 9
 $ cluster     : Named int [1:1108] 6 3 1 3 10 2 3 3 4 3 ...
  ..- attr(*, "names")= chr [1:1108] "able" "abnormal" "abrupt" "abstract" ...
 $ centers     : num [1:10, 1:300] -0.29695 -0.28824 0.06032 0.00418 -0.21374 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:10] "1" "2" "3" "4" ...
  .. ..$ : chr [1:300] "V1" "V2" "V3" "V4" ...
 $ totss       : num 37164
 $ withinss    : num [1:10] 2111 2234 5010 1788 1687 ...
 $ tot.withinss: num 31568
 $ betweenss   : num 5596
 $ size        : int [1:10] 80 83 163 65 71 150 90 119 58 229
 $ iter        : int 5
 $ ifault      : int 0
 - attr(*, "class")= chr "kmeans"
```

We are interested in the `cluster` element, which we extract and convert into a factor.

```
> fit <- as.factor(km.clustering$cluster)
```

Next, we append the factor to the original data frame with `cbind()`.

```
> newdata <- cbind(data, fit)
```

We run the `Rtsne()` function again.

```
> rtsne_out <- Rtsne(as.matrix(data), initial_dims=300)
```

We want to assign one color per class. Because there are ten classes, we need ten colors. The `rainbow()` function is perfect for this task because it specifies a range of hues, of which we select ten.

```
> colors <- rainbow(length(unique(newdata$fit)))
> colors
 [1] "#FF0000FF" "#FF9900FF" "#CCFF00FF" "#33FF00FF" "#00FF66FF" "#00FFFFFF" "#0066FFFF" "#3300FFFF"
 [9] "#CC00FFFF" "#FF0099FF"
```

We assign one color to each class with the `names()` function.

```
> names(colors) = unique(newdata$fit)
> colors
          6           3           1          10           2           4           7           5
"#FF0000FF" "#FF9900FF" "#CCFF00FF" "#33FF00FF" "#00FF66FF" "#00FFFFFF" "#0066FFFF" "#3300FFFF"
          9           8
"#CC00FFFF" "#FF0099FF"
```

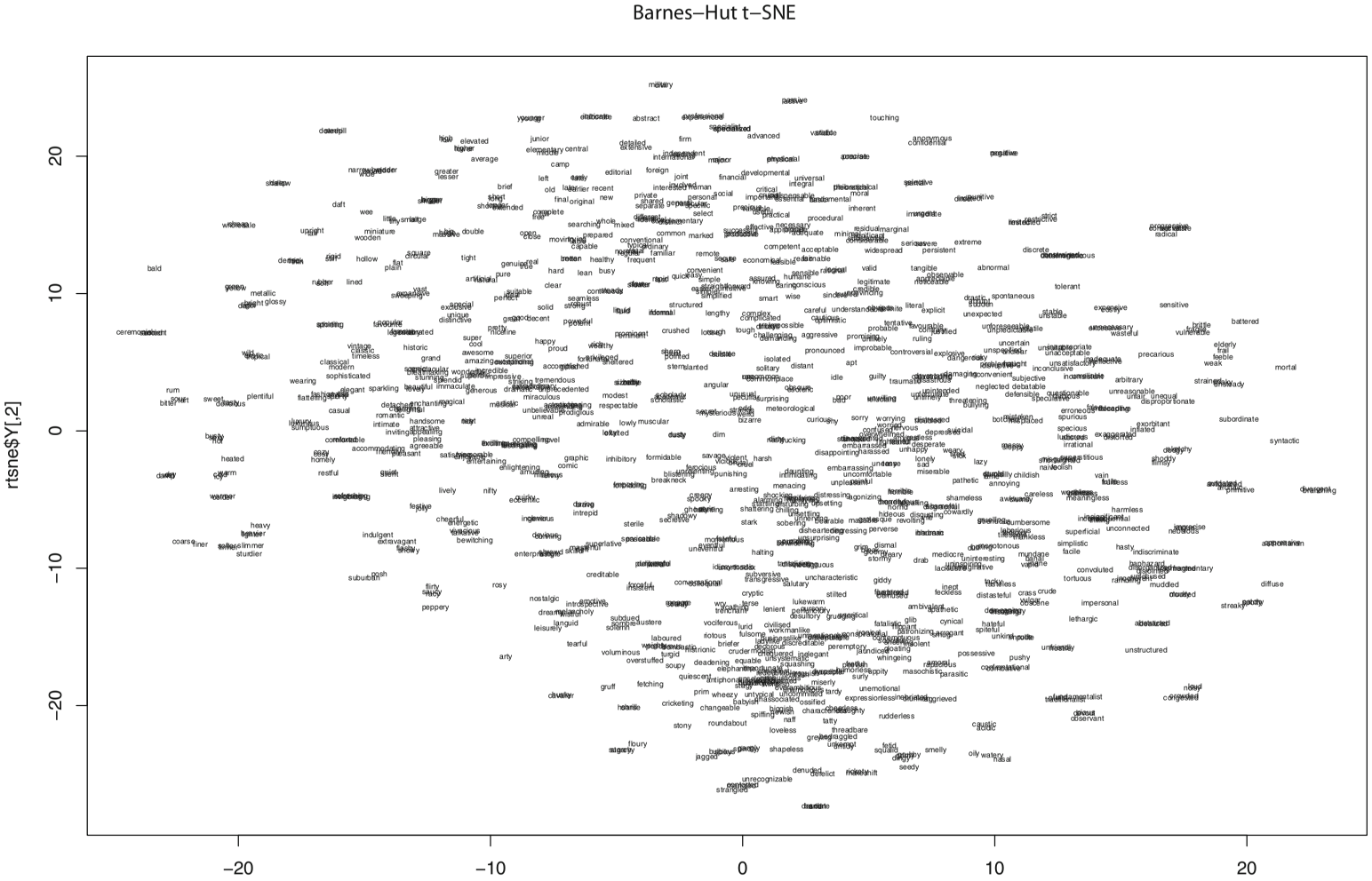


Fig. 10.7: A plot of adjectives based on 300-dimensional word vectors with t-SNE

Finally, we plot the adjectives, which are colored depending on the class that they belong to (Fig. 10.8). The *k*-means partitions do not necessarily correspond to how the adjectives are clustered by the t-SNE algorithm. This may be meaningful and worth interpreting.

```
> plot(rtsne_out$Y, t='n', main="Barnes-Hut t-SNE with k-means clustering")
> text(rtsne_out$Y, labels=rownames(data), col=colors[newdata$fit])
```

10.4 Correspondence Analysis

Correspondence analysis (henceforth CA) is a multifactorial method used to summarize and visualize large contingency tables. Its foundations were laid out by Benzécri (1973, 1984), and further described by Greenacre (2007) and Glynn (2014). It was popularized by Bourdieu (1979). It is now applied in corpus linguistics fairly regularly.

10.4.1 Principles of Correspondence Analysis

CA takes as input a matrix *M* of counts that consists of *I* individuals or observations (rows) and *J* variables (columns). This matrix is a contingency table. The method consists in reducing the dimensionality of *M* by calculating matrices between the rows and the columns. The output is then visualized on a two-dimensional map. The larger the distance between two rows and/or columns, the further apart the row or column coordinates will be on the map. The term "correspondence analysis" gets its name from what it aims to show, namely the correspondence between what the rows and the columns represent. Besides `FactoMineR`, there are several options for running CA in R: `ca` (Nenadic and Greenacre 2007), and `anacor` (de Leeuw and Mair 2009).

10.4.2 Case Study: General Extenders in the Speech of English Teenagers

Cheshire (2007) studies the use of general extenders in the speech of teenagers in England. General extenders are clause-final pragmatic particles appended to a word, a phrase, or a clause (2007, p. 156). Their basic pattern is a conjunction (*and* or *but*) followed by a noun phrase. Here are three examples:

(12) Steve works in the engineering office and has taken over some of the er you know purchasing function as well, like enquiries **and stuff**. (BNC–JP2)

(13) And I don't think we do it so much with the erm (pause) careers service training, but with the careers teachers often we've got them doing action plans **and things**. (BNC–G4X)

(14) I knew exactly where to stop it so that I could get off and go down the ladder, sneak a cup of tea **or something**. (BNC–FXV)

Cheshire recorded teenagers from three towns: Reading, Milton Keynes, and Hull. These towns are located in three distinct regional locations in England. In each town, the teenagers were grouped according

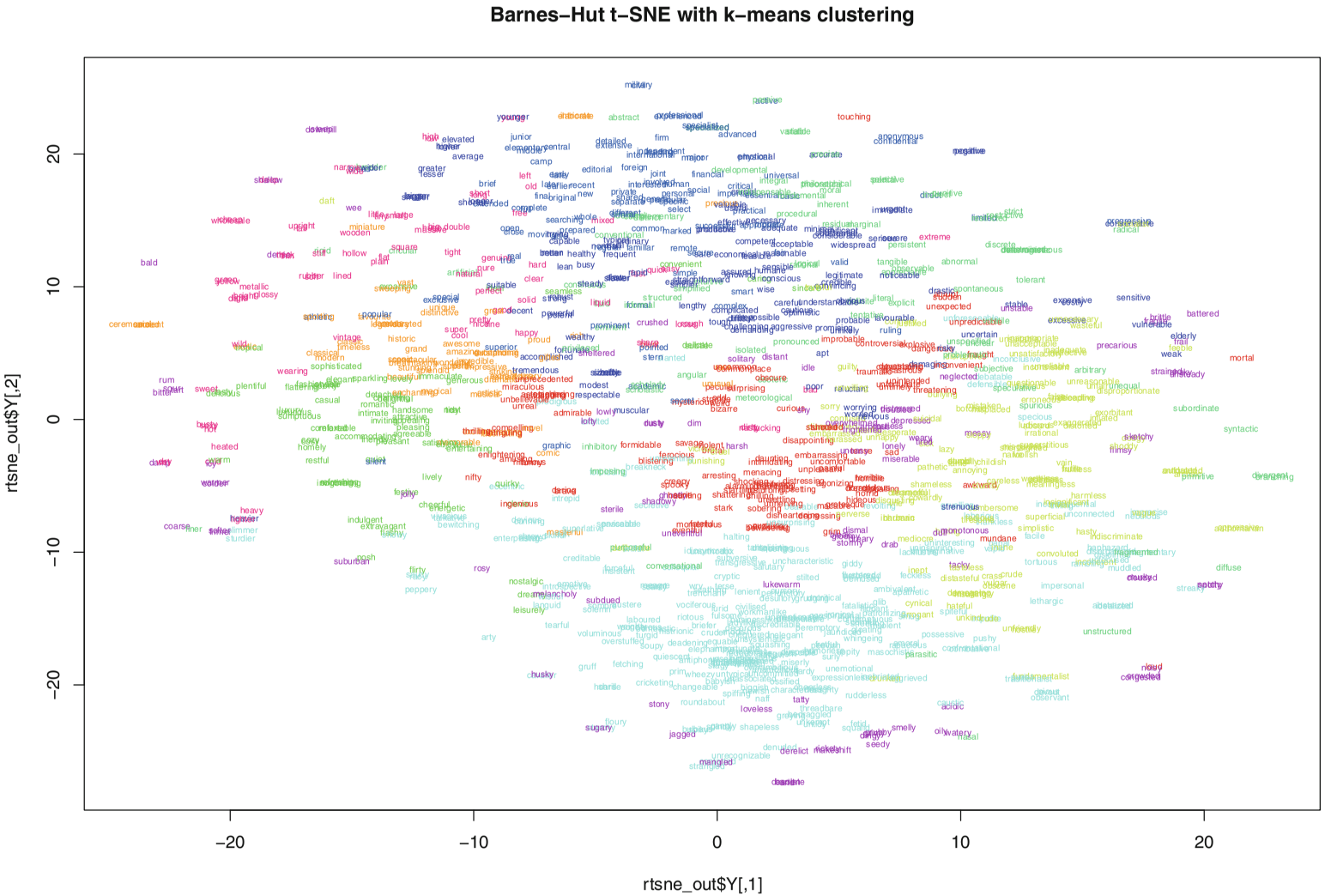


Fig. 10.8: A plot of adjectives based on 300-dimensional word vectors with t-SNE and *k*-means clustering

to gender and social class. After inspecting Tab. 10.3, which summarizes the distribution of the most frequent general extenders based on the social class of the informants, Cheshire finds that "there [is] a robust social-class distinction in the use of certain forms (...)" (2007, p. 164).

Table 10.3: The distribution of some general extenders in the speech of teenagers in three English towns (adapted from Cheshire 2007, p. 164)

forms	Reading		Milton Keynes		Hull	
	middle class	working class	middle class	working class	middle class	working class
and that	4	49	9	44	10	66
and all that	4	14	2	4	1	4
and stuff	36	6	45	5	62	18
and things	32	0	35	0	12	5
and everything	21	16	22	18	30	31
or something	72	20	30	17	23	3

Although modest in size, the contingency table is hard to synthesize with the naked eye. Another problem is that it does not display the marginal totals, i.e. the sum totals of the rows and columns. Any tendency we infer from raw frequencies may be flawed. This is where correspondence analysis steps in.

The aim of this part of Cheshire's study is to see if there are geographic/social differences across the data set and, if so, point out these differences. In fact, the point of making a table such as the above is twofold. You want to know which linguistic forms have the same regional and social profiles. You also want to know the linguistic profile of each region/social class with respect to general extenders.

In your `chap10` subfolder, you will find Tab. 10.3 in R data file format (`df_extenders.rds`). After clearing R's memory, load the `FactoMineR` library and import the data set into R.

```
> rm(list=ls(all=TRUE))
> library(FactoMineR)
> data <- readRDS("/CLSR/chap10/df_extenders.rds") # Windows
> data <- readRDS("/CLSR/chap10/df_extenders.rds") # Mac
```

The data set has been successfully imported as a data frame. With `str()`, inspect `data`.

```
> str(data)
'data.frame': 6 obs. of  6 variables:
 $ Reading_MC      : int  4 4 36 32 21 72
 $ Reading_WC      : int  49 14 6 0 16 20
 $ Milton_Keynes_MC: int  9 2 45 35 22 30
 $ Milton_Keynes_WC: int  44 4 5 0 18 17
 $ Hull_MC         : int  10 1 62 12 30 23
 $ Hull_WC         : int  66 4 18 5 31 3
```

It consists of six individuals (i.e. the six linguistic forms of the general extenders with one individual per row), and six variables (one variable per column). One minor difference with the original table is that the regional and social variables have been conflated. Thus, the column `Reading_WC` contains the contributions of teenagers of working class extraction from Reading whereas the column `Reading_MC` contains the contributions of teenagers from of middle class extraction from the same town.

The `CA()` function of the package `FactoMineR` is used to run the CA.

```
> ca.object <- CA(data)
```

By default, `CA()` outputs the plot that summarizes the contingency table (Fig. 10.9). The plot makes it easier to explore the meaningful tendencies in the data set. By default, the individuals are in blue and the variables in red.

Without the help of a CA graph, Cheshire (2007, p. 164) interprets the table as follows:

> Of the adjunctives, *and that* was preferred by the working-class speakers in all three towns, as was the less frequent *and all that*. The middle-class speakers, on the other hand, preferred *and stuff* and *and things* again in all three towns, though in Hull the middle-class adolescents used *and stuff* far more often than *and things* (the relatively high frequency of *and stuff* for the working-class group in Hull was due to just three speakers, one of whom was responsible for 10 of the 18 tokens).

The same divide between the linguistic practices of working class and middle class teenagers is observed in the graph. Indeed, the variables `Reading_MC`, `Miton_Keynes_MC`, and `Hull_MC` cluster in the left part of the plot whereas `Reading_WC`, `Miton_Keynes_WC`, and `Hull_WC` cluster in the right part of the plot. The general extenders *and that* and *and all that* do cluster together with the latter on the right. We also observe the surprising profile with respect to the three middle-class teenager from Hull, one of whom contributed 10 out of 18 occurrences of *and stuff* (see the lower left part of the plot). Cheshire further observes the following:

> There is a middle-class preference for *or something* in Reading and also in Hull, though in Hull the numbers of tokens are lower: however, this social class difference does not exist in the Milton Keynes data set.

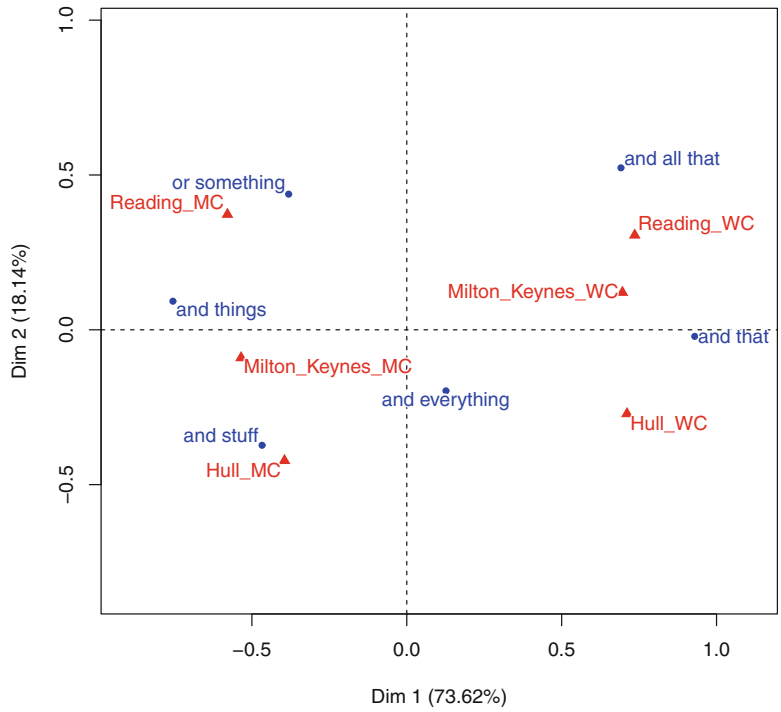


Fig. 10.9: CA biplot: a plane representation of individuals and variables in dimensions 1 and 2

Admittedly, *or something* is found in the left part of the plot—where the linguistic preferences of middle-class teenagers are found—and is closely associated with `Reading_MC`. Yet, the social class difference regarding this extender is not conspicuous for Hull in the graph because `Hull_MC` is not close to *or something*. Neither conspicuous is the absence of social class difference regarding Milton Keynes: `Milton_Keynes_MC` and *or something* are both on the left part of the plot. To understand why this is the case, let us wind back a little to understand what goes on statistically speaking when you run a CA.

10.4.3 How CA Works

Theoretically, the linguist makes no assumption as to what groupings should be found in the data set prior to running a CA. In practice, however, linguists construct a contingency table precisely because they expect to find a relationship between row and column variables. In fact, if the rows and the columns are independent, the operationalization of the research question has to be called into question. To make sure that the rows and columns are not independent, the χ^2 test is used. It tests the significance of the overall deviation of the table from the independence model. The test computes the contribution of each cell to χ^2 and sums up all contributions to obtain the χ^2 statistic.

The result is given by entering `ca.object`. The first part of the object states the following:

```
**Results of the Correspondence Analysis (CA)**
The row variable has 6 categories; the column variable has 6 categories
The chi square of independence between the two variables is equal to 384.1155
(p-value = 5.499003e-66 ).
```

Here, χ^2 has a high value and it is associated with very small p-value, implying that there is indeed a relationship between the row variables and the column variables. Yet, we should be wary of not using the magnitude of the χ^2 value to quantify the effect of the correlation between variables, because this value is dependent on the sample size. The data set does not meet one assumptions of the χ^2 test, which stipulates that 80% of the sample size should be greater than 5 (only about 67% of the sample is greater than 5). However, because CA is exploratory, it can be applied to tables even when the conditions of validity of the χ^2 statistic are not met (Greenacre 2007). Given that the p-value is very close to 0, the significance of the deviation of the table from independence is undeniable. In other words, the choice of a general extender and the locations and social classes of the informants are globally interdependent.

Central to CA is the concept of profile. To obtain the profile of a row, each cell is divided by its row total, as in Tab. 10.4. The last row in the table is the average row profile.

You obtain Tab. 10.4 thanks to Peter Dalgaard's function `prop.table()`, which converts the contingency table into a table of proportions based on marginal totals.

Table 10.4: The row profiles of Tab. 10.3

forms	Reading		Milton Keynes		Hull		row total
	middle class	working class	middle class	working class	middle class	working class	
and that	0.0220	0.2692	0.0495	0.2418	0.0549	0.3626	1
and all that	0.1379	0.4828	0.0690	0.1379	0.0345	0.1379	1
and stuff	0.2093	0.0349	0.2616	0.0291	0.3605	0.1047	1
and things	0.3810	0	0.4167	0	0.1429	0.0595	1
and everything	0.1522	0.1159	0.1594	0.1304	0.2174	0.2246	1
or something	0.4364	0.1212	0.1818	0.1030	0.1394	0.0182	1
column average	0.2195	0.1364	0.1857	0.1143	0.1792	0.1649	1

```
> data.col.totals <- rbind(data, apply(data[,1:6], 2, sum)) # sum up each column
> rownames(data.col.totals)[7] <- "column average" # add name to the last row
> data.col.totals <- as.matrix(data.col.totals) # convert to matrix
> row.profiles <-prop.table(data.col.totals, margin=1) # get proportions
> row.profiles <- cbind(row.profiles, rowSums(row.profiles)) # sum up each row
> colnames(row.profiles)[7] <- "row total" # add name to the last column
> row.profiles <- round(row.profiles, 4) # keep four decimal places
> row.profiles # display
               Reading_MC Reading_WC Milton_Keynes_MC Milton_Keynes_WC Hull_MC Hull_WC row total
and that           0.0220     0.2692           0.0495           0.2418  0.0549  0.3626         1
and all that       0.1379     0.4828           0.0690           0.1379  0.0345  0.1379         1
and stuff          0.2093     0.0349           0.2616           0.0291  0.3605  0.1047         1
and things         0.3810     0.0000           0.4167           0.0000  0.1429  0.0595         1
and everything     0.1522     0.1159           0.1594           0.1304  0.2174  0.2246         1
or something       0.4364     0.1212           0.1818           0.1030  0.1394  0.0182         1
column average     0.2195     0.1364           0.1857           0.1143  0.1792  0.1649         1
```

In a similar fashion, one obtains the profile of a column by dividing each column frequency by the column total (Tab. 10.5).

Table 10.5: The column profiles of Tab. 10.3

forms	Reading		Milton Keynes		Hull		row average
	middle class	working class	middle class	working class	middle class	working class	
and that	0.0237	0.4667	0.0629	0.5000	0.0725	0.5197	0.2364
and all that	0.0237	0.1333	0.0140	0.0455	0.0072	0.0315	0.0377
and stuff	0.2130	0.0571	0.3147	0.0568	0.4493	0.1417	0.2234
and things	0.1893	0	0.2448	0	0.0870	0.0394	0.1091
and everything	0.1243	0.1524	0.1538	0.2045	0.2174	0.2441	0.1792
or something	0.4260	0.1905	0.2098	0.1932	0.1667	0.0236	0.2143
column total	1	1	1	1	1	1	1

Again, you obtain Tab. 10.5 with `prop.table()`.

```
> data.row.totals <- cbind(data, apply(data[1:6,], 1, sum))
> colnames(data.row.totals)[7] <- "row average"
> col.profiles <-prop.table(as.matrix(data.row.totals), margin=2)
> col.profiles <- rbind(col.profiles, colSums(col.profiles))
> rownames(col.profiles)[7] <- "column total"
> col.profiles <- round(col.profiles, 4)
> col.profiles
               Reading_MC Reading_WC Milton_Keynes_MC Milton_Keynes_WC Hull_MC Hull_WC row average
and that           0.0237     0.4667           0.0629           0.5000  0.0725  0.5197      0.2364
and all that       0.0237     0.1333           0.0140           0.0455  0.0072  0.0315      0.0377
and stuff          0.2130     0.0571           0.3147           0.0568  0.4493  0.1417      0.2234
and things         0.1893     0.0000           0.2448           0.0000  0.0870  0.0394      0.1091
and everything     0.1243     0.1524           0.1538           0.2045  0.2174  0.2441      0.1792
or something       0.4260     0.1905           0.2098           0.1932  0.1667  0.0236      0.2143
column total       1.0000     1.0000           1.0000           1.0000  1.0000  1.0000      1.0000
```

Comparing row profiles to their average row profile leads to the same conclusions as comparing column profiles to their average column profile. For example, if we compare the profile of *and that* observed in the speech of middle class teenagers from Reading (0.0220) to the profile of all general extenders used by middle-class teenagers from Reading (0.2195) in Tab. 10.4, we obtain a ratio of $0.0220/0.2195 = 0.1002$. This implies that the use of *and that* among middle-class teenagers from Reading is well below the average profile of the general extenders among middle-class teenagers in Reading. In Tab. 10.5, the profile of *and that* in the same sociolinguistic context is 0.0237, whereas the average profile of *and that* is 0.2364. We obtain the same ratio as above, i.e. $0.0237/0.2364 = 0.1002$. The above profiles add up to 1. These relative frequencies have special geometric features, which CA converts into coordinates and visualizes as points on a map.

Let us now come back to Cheshire's claim that a middle-class preference is observed in Reading and Hull with regards to the use of *or something*, but not in Milton Keynes. Inspection of Tabs. 10.4 and 10.5 proves that she is right regarding Reading and Hull. However, even though the social class difference in Milton Keynes is not apparent in Tab. 10.5 (column profiles), it is more pronounced in Tab. 10.4 (row profiles). Furthermore, the difference between middle-class and working-class usage in Hull is not specific to *or something*. Bigger differences exist with respect to *and that* and *and stuff* in the row profiles and the column profiles. This explains why, although `Hull_MC` is on the left hand side of the biplot (like *or something*), the variable is not close to this extender. The two-dimensional map produced by CA goes a little farther than the eye can see.

On the one hand, representing both rows and columns on a single map is useful to capture the nature of the relationship between the row variables and the column variables. On the other hand, it does not say anything as to the intensity of the relationship. This intensity is measured with Cramér's V, introduced in Chap. 8.

```
> V <- sqrt(as.vector(chisq.test(data)$statistic)/(sum(data)*((min(ncol(data), nrow(data)))-1)))
> round(V, 3)
[1] 0.316
```

The intensity of the relationship is non negligible for this sort of data: $V = 0.316$. A score of 1 would be unrealistic as it would attest an exclusive association between the use of extenders and socio-geographic background. In CA, Cramér's V is assessed in the light of the variance of the table.

Distances between profiles are measured with inertia, which Greenacre (2007, p. 32) defines as "the weighted average of squared χ^2-distances between the row profiles and their average profile (similarly, between the column profiles and their average)". CA interprets inertia geometrically to assess how far row/column profiles are from their respective average profiles. It is with the total inertia of the table (Φ^2) that CA measures how much variance there is. Φ^2 is obtained by dividing the χ^2 statistic by the sample size.

```
> chisq <- chisq.test(data)
> inertia <- as.vector(chisq$statistic)/sum(chisq$observed); inertia
[1] 0.4988513
```

In Tab. 10.3, Φ^2 has a value of 0.5, which is not very high. Therefore, we can expect data points to be less spread out on the map than if Φ^2 were higher. This is because there are subtle differences between the data profiles.

Each column of the table contributes one dimension. CA decomposes Φ^2 along a few dimensions that concentrate as much information as possible. This is measured in terms of eigenvalues, which are components of the total inertia of the table (Φ^2).

```
> round(ca.object$eig, 2)
      eigenvalue percentage of variance cumulative percentage of variance
dim 1       0.37                 73.62                            73.62
dim 2       0.09                 18.14                            91.76
dim 3       0.03                  5.18                            96.94
dim 4       0.02                  3.05                            99.99
dim 5       0.00                  0.01                           100.00
```

Each dimension is associated with a percentage that reflects its contribution to Φ^2 (the larger the eigenvalue, the more the dimension contributes to Φ^2). In standard practice, the first few dimensions concentrate the largest eigenvalues. As evidenced by the output above and Fig. 10.10, the first two dimensions represents 91.76% of the variance of the table, that is to say most of the difference between the actual sample (the data table) and the theoretical sample under the independence hypothesis. Considering these two dimensions alone for interpretation is enough to capture the logic of Tab. 10.3.

```
> barplot(ca.object$eig[,2], names=paste("dimension", 1:nrow(ca.object$eig)),
+         xlab="dimensions", ylab="percentage of variance")
```

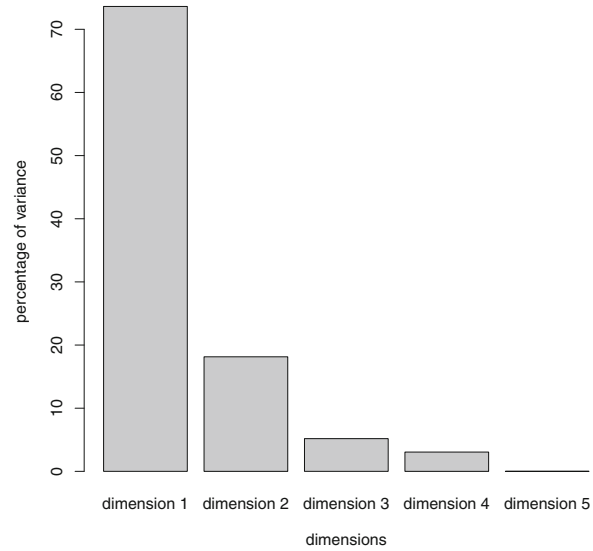


Fig. 10.10: A barplot showing the eigenvalues associated with each dimension

CA offers two indicators to facilitate the interpretation of biplots: the relative contributions of the data points to inertia and the quality of the projection of the data points onto the dimensions. Contributions are useful when there are many data points on the biplot. If a data point displays a minor contribution to a given

dimension, you must be careful not to overinterpret its position with respect to that dimension. Contributions are stored in `ca.object`. In one line of code, you can extract the contributions of rows and columns and combine them into one table using `rbind()`.

```
> contrib <- rbind(ca.object$row$contrib[,1:2], ca.object$col$contrib[,1:2])
```

With subsetting and `sort()`, you obtain the contributions to dimensions 1 and 2.

```
> sort(contrib[,1], decreasing=TRUE) # contributions of rows and columns to dim 1
        and that          Hull_WC       Reading_WC       Reading_MC       and things
      55.6242953       22.6977255       20.1086910       20.0324055       16.9323102
Milton_Keynes_WC Milton_Keynes_MC        and stuff     or something          Hull_MC
      15.0839737       14.4709370       13.2773723        8.4835958        7.6062673
    and all that   and everything
       4.9022705        0.7801559
> sort(contrib[,2], decreasing=TRUE) # contributions of rows and columns to dim 2
    or something          Hull_MC        and stuff       Reading_MC       Reading_WC
      45.4466061       35.3528820       34.3225961       33.6661156       14.0714353
         Hull_WC     and all that   and everything Milton_Keynes_WC Milton_Keynes_MC
      13.3971677       11.3867915        7.6981253        1.8360965        1.6763029
      and things         and that
       1.0280076        0.1178734
```

The extender *and that* and the variables `Hull_WC`, `Reading_WC`, and `Reading_MC` contribute the most to dimension 1 (along the horizontal axis in Fig. 10.9), unlike *or something*, `Hull_MC`, *and all that*, and *and everything*, which can be ignored here. Some of the data points, whose contribution to dimension 1 is negligible, contribute decisively to dimension 2 (along the vertical axis in Fig. 10.9): *or something*, `Hull_MC`, and *and stuff*.

The quality of the projection onto a dimension is measured as the percentage of inertia associated with this dimension.

```
> quality <- rbind(ca.object$row$cos2[,1:2], ca.object$col$cos2[,1:2])
> sort(quality[,1], decreasing=TRUE) # projection quality of rows and columns in dim 1
        and that Milton_Keynes_WC          Hull_WC Milton_Keynes_MC       and things
       0.9905390        0.9066550        0.8310275        0.8132865        0.7793381
      Reading_WC       Reading_MC        and stuff     and all that     or something
       0.7661048        0.6950972        0.5807247        0.4307165        0.4076990
         Hull_MC   and everything
       0.3997791        0.2689299
> sort(quality[,2], decreasing=TRUE) # projection quality of rows and columns in dim 2
  and everything     or something          Hull_MC        and stuff       Reading_MC
    0.6538582814     0.5381484992     0.4578406895     0.3698957352     0.2878369052
    and all that       Reading_WC          Hull_WC Milton_Keynes_WC Milton_Keynes_MC
    0.2465111508     0.1320942040     0.1208612359     0.0271933550     0.0232134877
      and things         and that
    0.0116586164     0.0005172065
```

Customarily, projection quality is used to select the dimension in which the individual or the variable is the most faithfully represented. For example, the contribution of `Reading_MC` is non-negligible in both dimensions, but the quality of its projection is at its highest in dimension 1. It is therefore with respect to the horizontal axis that this variable is to be interpreted in Fig. 10.9.

If you want to compare more than the first two dimensions, use the `axes` option.

```
> par(mfrow=c(1,2))
> plot(ca.object, title="")
> plot(ca.object, axes=2:3, title="")
```

In Fig. 10.11, the two biplots map the first three dimensions of the data. Dimension 1 is represented by the horizontal axis and dimension 2 by the vertical axis in the left plot. Dimension 2 is represented by the horizontal axis and dimension 3 by the vertical axis in the right plot.

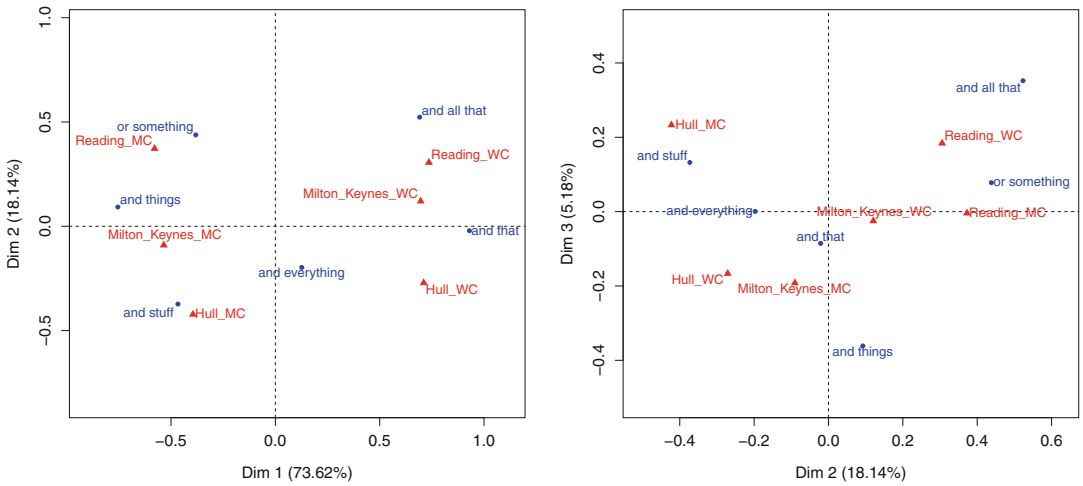


Fig. 10.11: Two CA biplots—*left*: dimensions 1 and 2; *right*: dimensions 2 and 3

Given that dimensions 1 and 2 account for most of Φ^2, let us focus on the left plot. The key to interpreting a CA biplot is to remember that, given the symmetric treatment of rows and columns, two rows or two columns have a similar profile if they are relatively close in the Euclidean space. The proximity of a particular row to a particular column indicates that this row has a particularly important weight in this column, and vice versa. As far as the column variables are concerned (in red), there is a clear divide between the use of extenders by middle-class (to the left) and working-class informants (to the right). As far as the row variables are concerned (in blue), the extenders favored by middle-class teenagers are *or something*, *and things*, and *and stuff*. The extenders favored by working-class teenagers are *and all that*, and *and that*. The general extender *and everything* stands in the middle. It does not contribute significantly to either dimension, but it is well projected along the vertical axis. This indicates that it is somewhat indifferent to socio-geographic variation. The vertical axis shows a difference between Reading and Milton Keynes in the upper part of the plot, and Hull at the bottom. Look at a map of England and you will see that this is not much of a surprise: Hull is in Yorkshire (northern England), whereas Reading and Milton Keynes are much further south.

10.4.4 Supplementary Variables

As with PCA, you can introduce illustrative elements in CA. These supplementary rows and/or columns help interpret the active rows and columns. As opposed to active elements, supplementary elements do not contribute to the construction of the dimensions. They can still be positioned on the map, but only after Φ^2 has been calculated with the active elements.

The column variables in Tab. 10.3 conflate the geographic origin and the social class of the informants, but what if you want to focus on either the geographic origin or the social background? What you can do is augment Tab. 10.3 with the totals for each town and each social class so as to obtain Tab. 10.6. In R, Tab. 10.6 is obtained by summing up the rows for each group of variables: Reading, Milton Keynes, Hull, middle class, and working class.

Table 10.6: Tab. 10.3 augmented with illustrative variables

forms	Reading		Milton Keynes		Hull		Reading	Milton_Keynes	Hull	MC	WC
	middle class	working class	middle class	working class	middle class	working class					
and that	4	49	9	44	10	66	53	53	76	23	159
and all that	4	14	2	4	1	4	18	6	5	7	22
and stuff	36	6	45	5	62	18	42	50	80	143	29
and things	32	0	35	0	12	5	32	35	17	79	5
and everything	21	16	22	18	30	31	37	40	61	73	65
or something	72	20	30	17	23	3	92	47	26	125	40

```
> # sum up the rows for Reading data and append the new column
> data.illust <- cbind(data, apply(data[1:6, 1:2], 1, sum))
> # add column name
> colnames(data.illust)[7] <- "Reading"
>
> # sum up the rows for Milton Keynes data and append the new column
> data.illust <- cbind(data.illust, apply(data.illust[1:6, 3:4], 1, sum))
> # add column name
> colnames(data.illust)[8] <- "Milton_Keynes"
>
> # sum up the rows for Hull data and append the new column
> data.illust <- cbind(data.illust, apply(data.illust[1:6, 5:6], 1, sum))
> # add column name
> colnames(data.illust)[9] <- "Hull"
>
> # sum up the rows for middle-class data and append the new column
> data.illust <- cbind(data.illust, apply(data.illust[1:6, c(1,3,5)], 1, sum))
> # add column name
> colnames(data.illust)[10] <- "MC"
>
> # sum up the rows for working-class data and append the new column
> data.illust <- cbind(data.illust, apply(data.illust[1:6, c(2,4,6)], 1, sum))
> # add column name
> colnames(data.illust)[11] <- "WC"
>
> #inspect
> str(data.illust)
'data.frame': 6 obs. of  11 variables:
 $ Reading_MC      : int  4 4 36 32 21 72
 $ Reading_WC      : int  49 14 6 0 16 20
 $ Milton_Keynes_MC: int  9 2 45 35 22 30
 $ Milton_Keynes_WC: int  44 4 5 0 18 17
 $ Hull_MC         : int  10 1 62 12 30 23
 $ Hull_WC         : int  66 4 18 5 31 3
 $ Reading         : int  53 18 42 32 37 92
 $ Milton_Keynes   : int  53 6 50 35 40 47
 $ Hull            : int  76 5 80 17 61 26
 $ MC              : int  23 7 143 79 73 125
 $ WC              : int  159 22 29 5 65 40
```

Your data frame now contains 11 columns, i.e. six active columns and five illustrative columns. As we run the function `CA()` on the data frame, we must declare the last five columns as illustrative with the `col.sup` argument.

```
> ca.object.2 <- CA(data.illust, col.sup=7:11)
```

Adding illustrative columns does not affect the χ^2 score and its associated p-value.

```
> ca.object.2
**Results of the Correspondence Analysis (CA)**
The row variable has  6  categories; the column variable has 6 categories
The chi square of independence between the two variables is equal to 384.1155 (p-value =  5.499003e-66 ).
*The results are available in the following objects:

   name              description
1  "$eig"            "eigenvalues"
2  "$col"            "results for the columns"
3  "$col$coord"      "coord. for the columns"
4  "$col$cos2"       "cos2 for the columns"
5  "$col$contrib"    "contributions of the columns"
6  "$row"            "results for the rows"
7  "$row$coord"      "coord. for the rows"
8  "$row$cos2"       "cos2 for the rows"
9  "$row$contrib"    "contributions of the rows"
10 "$col.sup$coord"  "coord. for supplementary columns"
11 "$col.sup$cos2"   "cos2 for supplementary columns"
12 "$call"           "summary called parameters"
13 "$call$marge.col" "weights of the columns"
14 "$call$marge.row" "weights of the rows"
```

If you generate the CA biplot based on `ca.object.2`, you get redundant projections with respect to the column variables.

```
> plot(ca.object.2)
```

Thanks to the `invisible` argument, you can choose which variables not to plot. In the code below, the active columns are made invisible.[8]

```
> plot(ca.object.2, invisible="col", title="")
```

Thanks to the illustrative variables, Fig. 10.12 makes the tendencies outlined above more conspicuous. Reading (upper part of the plot) and Hull (bottom part of the plot) have clearly distinct profiles, like middle class (left) and working class (right). The Milton Keynes variable is close to where the horizontal and vertical axes intersect. This point is where the average row profile and the average column profile coincide. It is also relatively to this intersection that all profiles on the map are measured. Because Milton Keynes is very close to the average profiles, no noteworthy tendency can be derived from this variable, contrary to what Tab. 10.3 might have led us to believe.

10.5 Multiple Correspondence Analysis

As its name indicates, multiple correspondence analysis, henceforth MCA, is an extension of CA. It takes as input tables with nominal categorical variables whereas CA takes as input tables with counts. This is because MCA was originally developed to explore the structure of surveys in which informants are asked to select an answer from a list of suggestions. For example, the question "According to you, which of these disciplines best describes the social sciences: literature, history, sociology, linguistics, anthropology?" requires informants to select one category.

When you extract data from a corpus, the table you obtain often takes the form of a table with individuals in the rows and categorical variables in the columns. Such is the case of the data frame that you compiled in

[8] Setting the values of `invisible` to `"row"`, `"row.sup"`, `"col"`, and `"col.sup"` will make the active rows, illustrative rows, active columns, and illustrative columns respectively invisible.

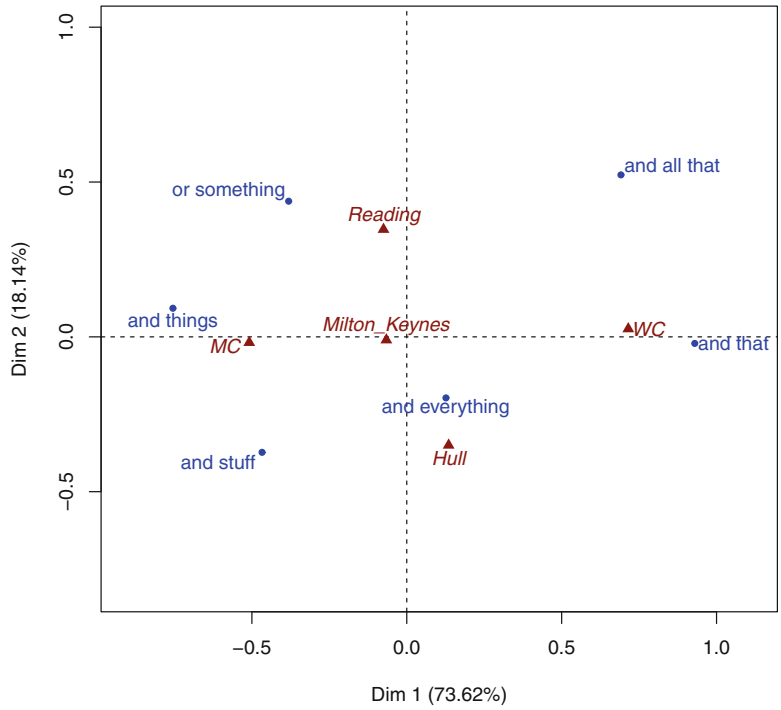


Fig. 10.12: CA biplot: a plane representation of active rows and illustrative columns

Sect. 5.3.3. Our starting point for MCA clustering is therefore a $I \times J$ data frame with I observations and J variables.

10.5.1 Principles of Multiple Correspondence Analysis

The general principles of CA outlined above also apply to MCA (Greenacre and Blasius 2006; Le Roux 2010). However, I find MCA to be more versatile than CA. For this reason, MCA is perhaps more complex than CA. This is a reasonable price to pay for the subtle interpretations that you will make from a study based on MCA.

In MCA, you may choose to focus on the individuals, the variables, or the categories of these variables. The categories are important because they provide a link between the individuals and the variables. If you choose to focus on individuals, you want to examine the proximities between them in terms of the variables and/or the categories that characterize them. The more categories individuals have in common, the more similar their profiles and the closer they will be on a map. You may also want to summarize the relationships between variables (this is what we shall do below). When you focus on categories, the more individuals two categories have in common, the closer they are.

10.5.2 Case Study: Predeterminer vs. Preadjectival Uses of Quite *and* Rather

MCA is illustrated using a case study based on Desagulier (2015). The purpose of this paper was to tease out differences between *quite* and *rather*, the underlying assumption being that these two adverbs are near-synonyms. Although *quite* and *rather* can modify other adverbs (*quite frankly*, *rather desperately*), NPs (*quite a sight*, *rather a shock*), or even VPs (*I quite understand*, *I rather enjoyed it*), the investigation was restricted to the far more frequent contexts where these intensifiers are used as degree modifiers of adjectives, as in (15) and (16).

(15) We are *quite different* people now, we need different things. (BNC–CEX)

(16) As it happens, there is a *rather nice* place for sale, not too far from London. (BNC–H9V)

When *quite* and *rather* modify attributive adjectives, they can occur in predeterminer position, a behavior that we do not see with other intensifiers.

(17) a. That has proved to be *a quite difficult* question to answer. (preadjectival)
b. That has proved to be *quite a difficult* question to answer. (predeterminer)
a. That is *a rather difficult* question to answer. (preadjectival)
b. That is *rather a difficult* question to answer. (predeterminer)
a. I know it is *a fairly difficult* question. (preadjectival)
b. ??I know it's *fairly a difficult* question. (predeterminer)

The study was partly prompted by the popular belief that the predeterminer pattern is significantly associated with written language (and the formality traditionally associated with it).

To remain within the strict limits of the preadjectival vs. predeterminer alternation, I kept only those patterns that involved both alternants, namely <*a(n) quite/rather* ADJ NP> and <*quite/rather a(n)* ADJ NP>. Following the same logic, I ignored the cases where the adjective was in predicative position. To see how the preadjectival vs. predeterminer alternation behaves in the light of contextual variables, I extracted contextual tags of the XML edition of the British National Corpus and compiled a dataset with three levels of contextual information for each observation: text mode, text type, and information regarding each text. I obtained a data frame that consists of about 3100 observations and 6 categorical variables:

- construction (predeterminer vs. preadjectival),
- intensifier (*quite* and *rather*),
- text mode (written vs. spoken),
- text type (othersp, news, fiction, etc.),
- text information (W fict prose, W ac:humanities arts, S conv, etc.),
- semantic classes of adjectives.

The dataset was submitted to multiple correspondence analysis.

In your `chap10` subfolder, you will find a miniature version of the data frame that I used in the above-mentioned paper: `quite.rather.rds`. It has been compiled from the BNC Baby instead of the full BNC (XML Edition).

After clearing R's memory, load the data frame and inspect it.

```
> rm(list=ls(all=T))
> df.quite.rather <- readRDS("/CLSR/chap10/quite.rather.rds")
> str(df.quite.rather)
'data.frame': 163 obs. of  9 variables:
 $ corpus_file : Factor w/ 69 levels "A1F.xml","A1N.xml",..: 57 38 6 32 40 13 43 43 27 39 ...
 $ construction: Factor w/ 2 levels "PREADJECTIVAL",..: 2 1 2 2 2 1 1 1 1 1 ...
 $ intensifier : Factor w/ 2 levels "QUITE","RATHER": 1 2 1 1 2 1 1 1 2 2 ...
 $ text_mode   : Factor w/ 2 levels "SPOKEN","WRITTEN": 1 2 2 2 2 2 2 2 2 2 ...
 $ text_type   : Factor w/ 4 levels "ACPROSE","CONVRSN",..: 2 3 4 3 3 4 3 3 1 3 ...
 $ text_info   : Factor w/ 18 levels "S conv","W ac:humanities arts",..: 1 6 13 6 6 10 6 6 2 6 ...
 $ match       : Factor w/ 150 levels "a quite different class ",..: 133 15 69 76 147 1 2 3 18 19 ...
 $ adjective   : Factor w/ 89 levels "able","aggressive",..: 1 10 10 19 69 21 21 21 21 21 ...
 $ sem_class   : Factor w/ 18 levels "Able/intelligent",..: 1 1 1 1 1 2 2 2 2 2 ...
```

MCA is very sensitive to the number of categories per variable. In R, this is visible when you inspect the structure of the data frame with `str()`: `Factor w/ n levels`. If there are too many levels, it will affect the inertia of the table and therefore its breaking down into eigenvalues. You will therefore need to inspect a lot more than just the first two dimensions (which is common in MCA anyway).

Furthermore, if some factor levels concern too small a number of individuals, a large inertia will result and these factors will be overrepresented. Such is the case of some categories of the variables `text_info`.

```
> sort(table(df.quite.rather$text_info), decreasing=TRUE)

                       S conv                   W fict prose              W ac:nat science
                           57                             47                            11
        W newsp other: report          W ac:humanities arts              W ac:soc science
                            8                              7                             7
           W ac:polit law edu      W newsp brdsht nat: misc           W newsp other: arts
                            6                              5                             4
     W newsp brdsht nat: arts  W newsp brdsht nat: commerce W newsp brdsht nat: editorial
                            3                              1                             1
   W newsp brdsht nat: report   W newsp brdsht nat: science    W newsp brdsht nat: social
                            1                              1                             1
   W newsp brdsht nat: sports        W newsp other: science               W newsp tabloid
                            1                              1                             1
```

Several of the categories prefixed by `W newsp brdsht` characterize very few observations (and often just one). We should therefore group these rare categories into larger ones. The function `levels()` allows you to access each level of the variable `text_info`.

```
> levels(df.quite.rather$text_info)
 [1] "S conv"                          "W ac:humanities arts"          "W ac:nat science"
 [4] "W ac:polit law edu"              "W ac:soc science"              "W fict prose"
 [7] "W newsp brdsht nat: arts"        "W newsp brdsht nat: commerce"  "W newsp brdsht nat: editorial"
[10] "W newsp brdsht nat: misc"        "W newsp brdsht nat: report"    "W newsp brdsht nat: science"
[13] "W newsp brdsht nat: social"      "W newsp brdsht nat: sports"    "W newsp other: arts"
[16] "W newsp other: report"           "W newsp other: science"        "W newsp tabloid"
```

You may now replace the rare categories with a higher-order category using the indices provided by `levels()` above.

```
> levels(df.quite.rather$text_info)[7:14] <- "W newsp brdsht nat"
> levels(df.quite.rather$text_info)[8:11] <- "W newsp other"
```

Regarding the variable `sem_class`, the category `Comparison_Usual` concerns only one observation.

```
> sort(table(df.quite.rather$sem_class), decreasing=TRUE)

                Evaluation_Good            Comparison_Different            Size_Big_Large_Heavy
                             37                              19                              19
                 Evaluation_Bad            Time_period_long_old                 Psych_state_neg
                             17                              14                              13
```

```
Phys_percept_Beautiful_pleasant          Comparison_Unusual                Size_Small_Short
                              9                           7                               6
            Able/intelligent                        Easy                  Entire_maximum
                              5                           3                               3
         Comparison_Similar                Definite_Detailed                      Difficult
                              2                           2                               2
   Phys_percept_Ugly_unpleasant        Time_period_short_young                Comparison_Usual
                              2                           2                               1
```

We group it with `Comparison_Similar`.

```
> levels(df.quite.rather$sem_class)[5] <- "Comparison_Similar"
```

In this first MCA, we restrict the variables to those listed above (`construction`, `intensifier`, `text_mode`, `text_type`, and `text_info`) but leave aside `sem_class` so as not to clutter the graph.

```
> df.mca <- df.quite.rather[, c(2:6)]
> str(df.mca)
'data.frame': 163 obs. of  5 variables:
 $ construction: Factor w/ 2 levels "PREADJECTIVAL",..: 2 1 2 2 2 1 1 1 1 1 ...
 $ intensifier : Factor w/ 2 levels "QUITE","RATHER": 1 2 1 1 2 1 1 1 2 2 ...
 $ text_mode   : Factor w/ 2 levels "SPOKEN","WRITTEN": 1 2 2 2 2 2 2 2 2 2 ...
 $ text_type   : Factor w/ 4 levels "ACPROSE","CONVRSN",..: 2 3 4 3 3 4 3 3 1 3 ...
 $ text_info   : Factor w/ 8 levels "S conv","W ac:humanities arts",..: 1 6 7 6 6 7 6 6 2 6 ...
```

All the variables are declared as active except `text_info`, which we declare as illustrative (or "supplementary qualitative variable") thanks to the argument `quali.sup` of the `MCA()` function.

```
> library(FactoMineR)
> mca.object <- MCA(df.mca, quali.sup=5, graph=FALSE)
```

By default, `MCA()` generates several graphs: a graph of variables (including supplementary variables), a graph of categories, and a graph of individuals.[9] To prevent `MCA()` from plotting these graphs, switch `graph` to `FALSE`. We can plot the graph(s) that we want later with the `plot()` or `plot.MCA()` functions, which offer more plotting options.

As in CA, each dimension is associated with a percentage that reflects its contribution to Φ^2. To determine the number of representative dimensions, examine the cumulative percentage of variance.

```
> round(mca.object$eig, 2)
       eigenvalue percentage of variance cumulative percentage of variance
dim 1        0.72                 47.97                            47.97
dim 2        0.31                 20.44                            68.41
dim 3        0.25                 16.68                            85.09
dim 4        0.15                  9.87                            94.96
dim 5        0.08                  5.04                           100.00
dim 6        0.00                  0.00                           100.00
```

You may also represent the results graphically with a barplot (Fig. 10.13).

```
> barplot(mca.object$eig[,2], names=paste("dimension", 1:nrow(mca.object$eig)))
```

The first two dimensions account for 68.41% of Φ^2. The cumulative percentage of the first two dimensions is usually much lower than what you witness in CA. Here, we are lucky because the percentage is unusually high.[10] The contribution of dimension 2 is not much smaller than the contribution of dimension 3 (dim 2: 20.44; dim 3: 16.68), which is why examining the latter at some stage is not a bad idea.

[9] A graph dedicated to quantitative variables will also be plotted if your data frame contains such variables.

[10] I designed the data set on purpose.

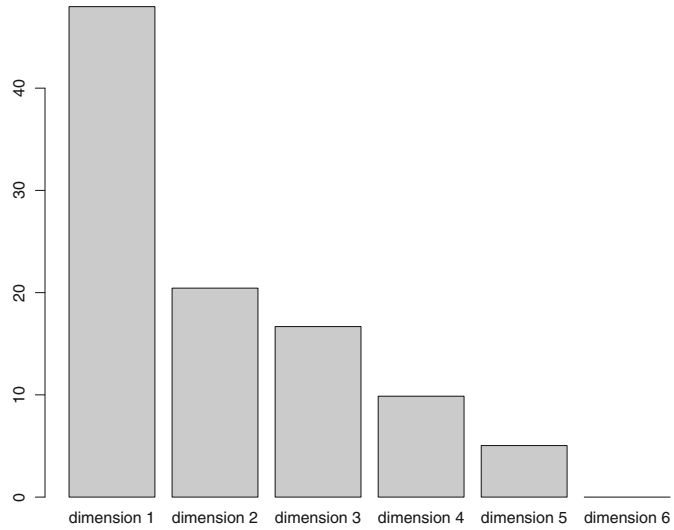


Fig. 10.13: A barplot showing the eigenvalues associated with each dimension

Because the data set consists of a limited number of categories we can represent them simultaneously. The individuals are of little importance (they are just distinct corpus-based observations) so we make them invisible by specifying `invisible="ind"`. If we do not, each observation will appear on the map as a number, which will clutter the graph.[11]

```
> plot.MCA(mca.object, invisible="ind", autoLab="yes", shadow=TRUE, habillage="quali",
+          cex=0.6, title="")
```

Fig. 10.14 maps the first two dimensions of the data. It is a good idea to back up your interpretation of a MCA biplot with numerical descriptors. You can display them thanks to the `FactoMineR`-specific function `dimdesc()`.

```
> descriptors <- dimdesc(mca.object, axes=c(1,2))
```

Two kinds of descriptors are found for each dimension.

```
> descriptors$`Dim 1`$category
                        Estimate      p.value
WRITTEN                0.7854541 8.669160e-55
PREADJECTIVAL          0.7062451 1.396927e-37
RATHER                 0.6904487 2.544149e-35
ACPROSE                0.7306224 9.514332e-12
W fict prose           0.1270017 2.959679e-08
FICTION                0.4096808 2.959679e-08
W ac:humanities arts   0.6825574 3.327880e-04
W ac:nat science       0.3732726 1.042961e-03
W ac:polit law edu     0.4408545 1.026816e-02
W ac:soc science       0.3367449 1.443907e-02
```

[11] The arguments `autolab` and `shadow` are specific to `FactoMineR`: they prevent data points and their respective labels from being overplotted. The argument `habillage` is set to `"quali"`, which means that the categories are assigned one color depending on what variable they belong to. The argument `cex` is used to specify the size of the plotted labels: `0.6` means the font size is 60% of the default size. The default main title of the plot is removed with `title=""`.

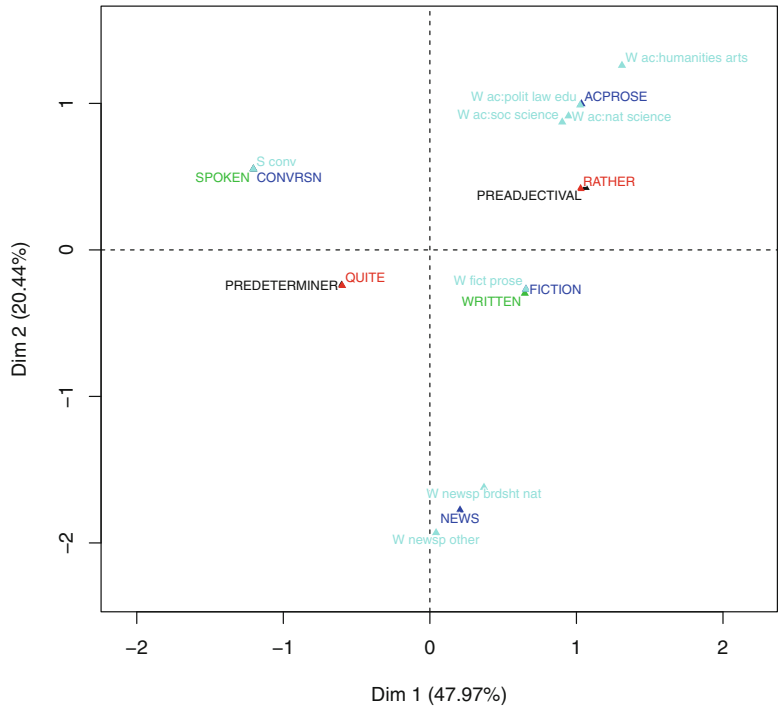


Fig. 10.14: MCA biplot: a simultaneous representation of preadjectival vs. predeterminer syntax (active, in *black*), intensifiers (active, in *red*), text modes (active, in *green*), text types (active, in *blue*), and text information (illustrative, in *cyan*)

```
QUITE                -0.6904487 2.544149e-35
PREDETERMINER        -0.7062451 1.396927e-37
S conv               -1.4508915 8.669160e-55
CONVRSN              -1.1682124 8.669160e-55
SPOKEN               -0.7854541 8.669160e-55
```

We observe a strong correlation between (a) *quite* and the predeterminer construction in the lower left corner of the plot, and (b) between *rather* and the preadjectival construction in the upper right corner. Along the horizontal axis (dimension 1), predeterminer *quite* is correlated with the `SPOKEN` category of the `text_mode` variable, and more specifically with the catogory `CONVRSN` (=conversation) of the `text_type` variable. In contrast, along the same axis, preadjectival *rather* is correlated with the `WRITTEN` modality of `text_mode` and all the modalies of `text_type`: `ACPROSE`, `FICTION`, and `NEWS`.[12]

Regarding dimension 2, we should be wary of concluding that predeterminer *quite* occurs exclusively in spoken contexts and preadjectival *rather* in written contexts. Indeed, pay attention to the first six lines and the last seven lines of the table below, which displays the descriptors for the categories in dimension 2.

```
> descriptors$`Dim 2`$category
                      Estimate      p.value
ACPROSE              0.62071175 6.466665e-11
```

[12] Although far from `RATHER`, `NEWS` is also found in the right part of the plot.

```
S conv                0.25249883 8.508201e-08
CONVRSN               0.37412124 8.508201e-08
SPOKEN                0.23486899 8.508201e-08
PREADJECTIVAL         0.18474516 3.000038e-05
RATHER                0.18313246 3.315603e-05
W ac:humanities arts  0.64419427 5.784886e-04
W ac:nat science      0.45318397 1.555289e-03
W ac:polit law edu    0.49442785 1.344386e-02
W ac:soc science      0.43011700 1.818796e-02
W fict prose         -0.20173368 2.906486e-02
FICTION              -0.08011128 2.906486e-02
QUITE                -0.18313246 3.315603e-05
PREDETERMINER        -0.18474516 3.000038e-05
WRITTEN              -0.23486899 8.508201e-08
W newsp brdsht nat   -0.95063807 1.507307e-11
W newsp other        -1.12205017 8.835144e-17
NEWS                 -0.91472171 5.842538e-39
```

Along the vertical axis (dimension 2), predeterminer *quite* is found in two written contexts: `FICTION` and `NEWS`. Along the same axis, preadjectival *rather* is found in one written context (`ACPROSE`) but also one spoken context (`CONVRSN`). MCA shows that the divide between the uses of *quite* and *rather* is a matter of degree. Given the greater contribution of dimension 1 to Φ^2, we can say that *quite* occurs preferentially in predeterminer position. This pattern tends to appear both in the written and spoken components of the corpus, with a preference for the latter. *Rather* occurs preferentially in pre-adjectival position. This pattern tends to appear both in the written and spoken components of the corpus, with a preference for the former. Contrary to popular belief, we find that the predeterminer construction is not exclusively associated with written language in the BNC Baby. Of course, these conclusions are valid for the corpus only. Whether they can be extended beyond the corpus cannot be determined with MCA because this method is only exploratory.

10.5.3 Confidence Ellipses

One asset of `FactoMineR` with respect to the other available packages that can do MCA is the possibility to draw confidence ellipses around the categories thanks to the function `plotellipses()`. The argument `keepvar` selects the number of categories that you want to plot. By default, the confidence level is set to 0.95. You can modify it by changing the value of the `level` argument. Finally, `magnify` controls the magnification of level names.

In the line of code below, `keepvar` selects the first four variables by means of the columns indices: `construction`, `intensifier`, `text_mode`, and `text_type`. These four categories are the first four columns of the data table.

```
> plotellipses(mca.object, keepvar=c(1:4), magnify = 1)
```

In Fig. 10.15, the ellipses do not overlap, which means the categories are significantly distinct. Let us now plot the confidence ellipses of the categorical variable `text_info`.

```
> plotellipses(mca.object, keepvar=5, magnify = 1)
```

In Fig. 10.16, the ellipses of two sets of categories within `text_info` overlap: `W newsp brdsht nat` and `W newsp other` on the one hand, and `W ac:humanities arts`, `W ac:nat science`, `W ac:polit law edu`, and `W ac:soc science` on the other hand. This means that the categories within each set are not significantly distinct.

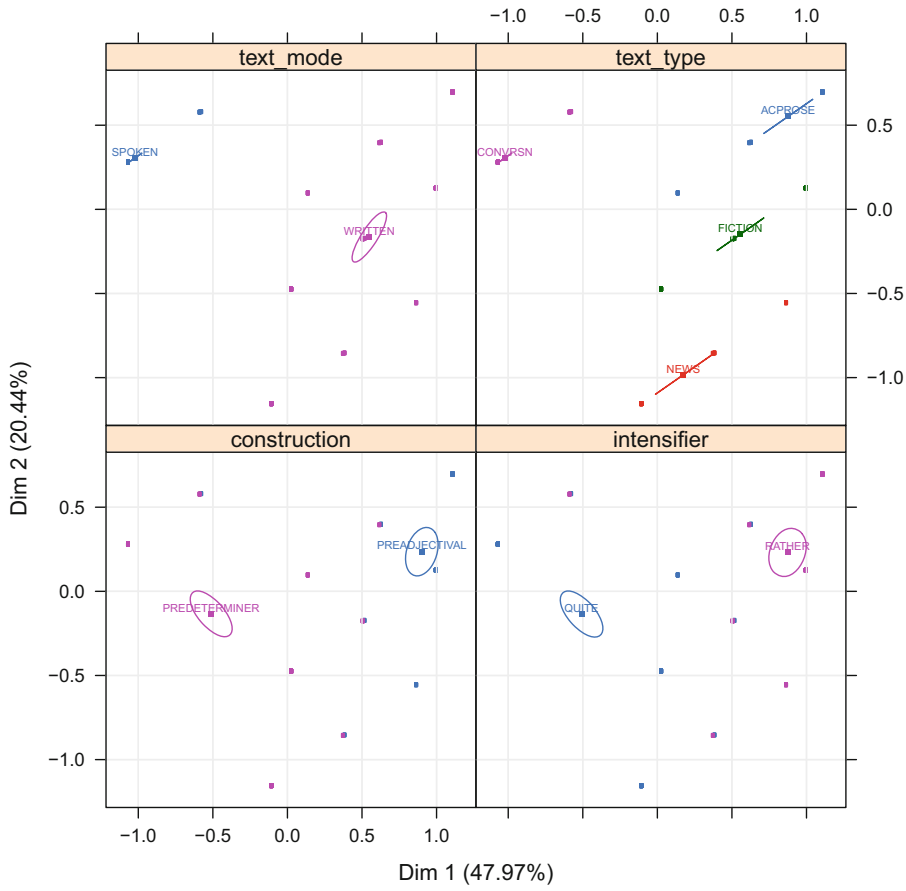


Fig. 10.15: MCA: confidence ellipses for four variables

10.5.4 Beyond MCA

Some statisticians do not like the fact that in the case of data frames with many categorical variables and categories, you have to inspect a large number of dimensions. Joint correspondence analysis has been suggested as a simple alternative (Camiz and Gomes 2013).

10.6 Hierarchical Cluster Analysis

Hierarchical cluster analysis (henceforth HCA) clusters individuals on the basis of the distance between them (Everitt et al. 2011, Sect. 4.2). The distance between individuals depends on the variables that characterize them. HCA has been used widely in linguistics. Some relevant references include Divjak and Fieller (2014),

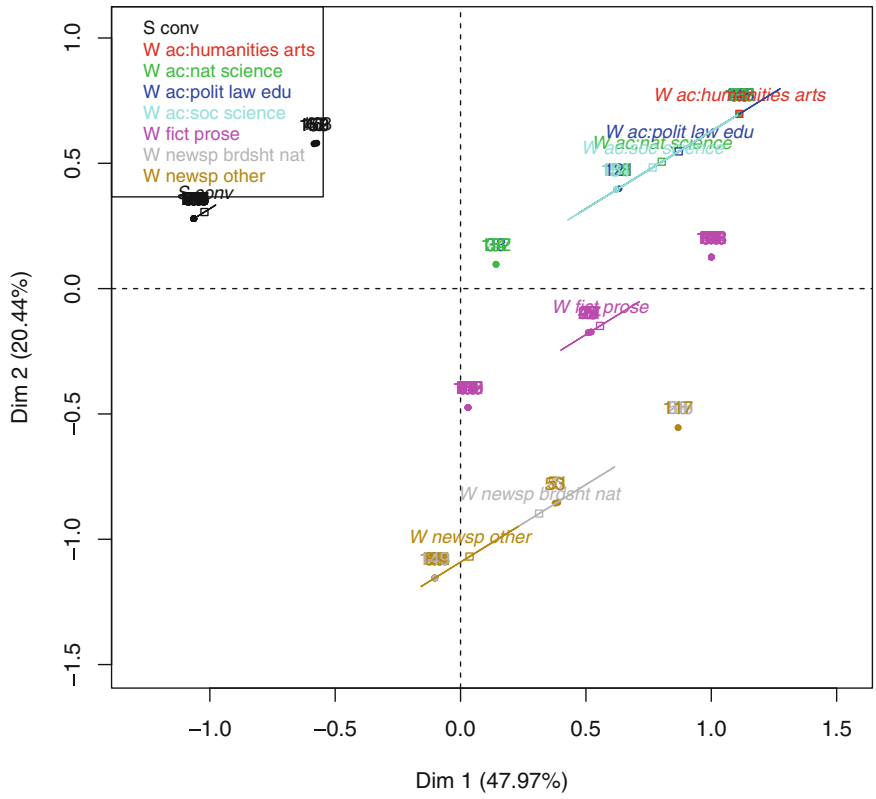


Fig. 10.16: MCA: confidence ellipses around the categories of `text_info`

Divjak and Gries (2006, 2008), and Gries and Stefanowitsch (2010). HCA is available from several R packages: `hclust`, `diana`, `cluster` (`agnes()`), or `pvclust`.

10.6.1 The Principles of Hierarchical Cluster Analysis

HCA takes as input a data table T of I individuals or observations and K variables. The response variable may be discrete (integers), continuous (real numbers), or both.[13] HCA converts the table into a distance object to which an amalgamation rule is applied. The amalgamation rule specifies how the elements in the distance object are assembled into clusters.

[13] See Husson et al. (2011) to learn how to integrate qualitative variables.

10.6.2 Case Study: Clustering English Intensifiers

I illustrate HCA with a case study from Desagulier (2014).

```
> rm(list=ls(all=TRUE))
> data <- readRDS("C:/CLSR/chap10/hca_intensifiers.rds") # Windows
> data <- readRDS("/CLSR/chap10/hca_intensifiers.rds") # Mac
```

The data set was compiled to see how 23 English intensifiers cluster on the basis of their most associated adjectives. To obtain a list of the 23 intensifiers, you can use the `rownames()` function because there is one row per intensifier.

```
> rownames(data)
 [1] "a_bit"       "a_little"    "absolutely"  "almost"      "awfully"     "completely"
 [7] "entirely"    "extremely"   "fairly"      "frightfully" "highly"      "jolly"
[13] "most"        "perfectly"   "pretty"      "quite"       "rather"      "slightly"
[19] "somewhat"    "terribly"    "totally"     "utterly"     "very"
```

Let me explain how the data set was compiled. For each of the 23 adverbs listed above, I first extracted all adjectival collocates from the Corpus of Contemporary American English (Davies 2008–2012), amounting to 432 adjective types and 316159 co-occurrence tokens. Then, I conducted a collexeme analysis for each of the 23 degree modifiers. To reduce the data set to manageable proportions, the 35 most attracted adjectives were selected on the basis of their respective collostruction strengths, yielding a 23-by-432 cooccurrence table (or contingency table) containing the frequency of adverb-adjective pair types.

The contingency table must be converted into a distance object. Technically, this distance object is a dissimilarity matrix. A dissimilarity matrix is similar to the driving distance tables that you used to find in road atlases (Tab. 10.7). Because the matrix is symmetric, it is divided into two parts (two triangles) on either side of the diagonal of null distances between the same cities. Only one triangle is needed. You obtain the

Table 10.7: Matrix of driving distances between five US cities

	Dallas	Miami	New York	San Francisco	Seattle
Dallas	0	1312.1	1548.5	1731.4	2109.1
Miami	1312.1	0	1283.3	3114.3	3296.9
New York	1548.5	1283.3	0	2568.6	2404.6
San Francisco	1731.4	3114.3	2568.6	0	807
Seattle	2109.1	3296.9	2404.6	807	0

dissimilarity matrix by converting the contingency table into a table of distances with a user-defined distance measure. When the variables are ratio-scaled, you can choose from several distance measures: Euclidean, City-Block/Manhattan, correlation, Pearson, Canberra, etc. For reasons of space, I cannot discuss the reasons why one should prefer a distance measure over another.[14] I find the Canberra distance metric to best handle the relatively large number of empty occurrences that we typically obtain in linguistic data. We use the `dist()` function. The first argument is the data table. The second argument is the distance metric (here, Canberra). The third argument (`diag`) lets you decide if you want R to print the diagonal of the distance

[14] A description of some distance measures can be found in Gries (2010, pp. 313–316). See also Divjak and Gries (2006).

object. The fourth argument (`upper`) lets you decide if you want R to print the upper triangle of the distance object.

```
> dist.object <- dist(data, method="canberra", diag=T, upper=T)
```

The function `dist()` outputs a distance object, which has a class of its own (it is not exactly a matrix object).

```
> class(dist.object)
[1] "dist"
```

If you want to display the whole distance matrix, type `dist.matrix`. If you want to display only a snapshot (e.g. the first five rows and the first five columns), first convert the distance object into a matrix, and then subset the matrix.

```
> dist.matrix <- as.matrix(dist.object)
> dist.matrix[1:5, 1:5]
              a_bit a_little absolutely   almost  awfully
a_bit        0.0000 372.1788   432.0000 432.0000 418.8591
a_little   372.1788   0.0000   429.4200 426.3215 422.8858
absolutely 432.0000 429.4200     0.0000 421.9386 432.0000
almost     432.0000 426.3215   421.9386   0.0000 432.0000
awfully    418.8591 422.8858   432.0000 432.0000   0.0000
```

The diagonal of `0` values separates the upper and lower triangles, as expected from a distance matrix.

Finally, use the `hclust()` function to apply an amalgamation rule that specifies how the elements in the matrix are clustered. We amalgamate the clusters with Ward's method (Ward 1963), which evaluates the distances between clusters using an analysis of variance. Ward's method is the most widely used amalgamation rule because it has the advantage of generating clusters of moderate size. We specify `method="ward.D"`.

```
> clusters <- hclust(dist.object, method="ward.D")
```

Fig. 10.17 shows the resulting dendrogram, which you obtain with `plot()`.

```
> plot(clusters, sub="(Canberra, Ward)")
```

The `sub` argument allows you to annotate your dendrogram. I recommend that you specify the distance metric and the amalgamation rule.

10.6.3 Cluster Classes

Two more function help you spot and interpret the relevant clusters. Thanks to the `rect.hclust()` function, you can assign groups. The code below draws red rectangles around the branches of the dendrogram, highlighting six groups of clusters (Fig. 10.18).

```
> rect.hclust(clusters, 6)
```

The `cutree()` function lists the individuals that belong to each group. Each group is assigned a digit.

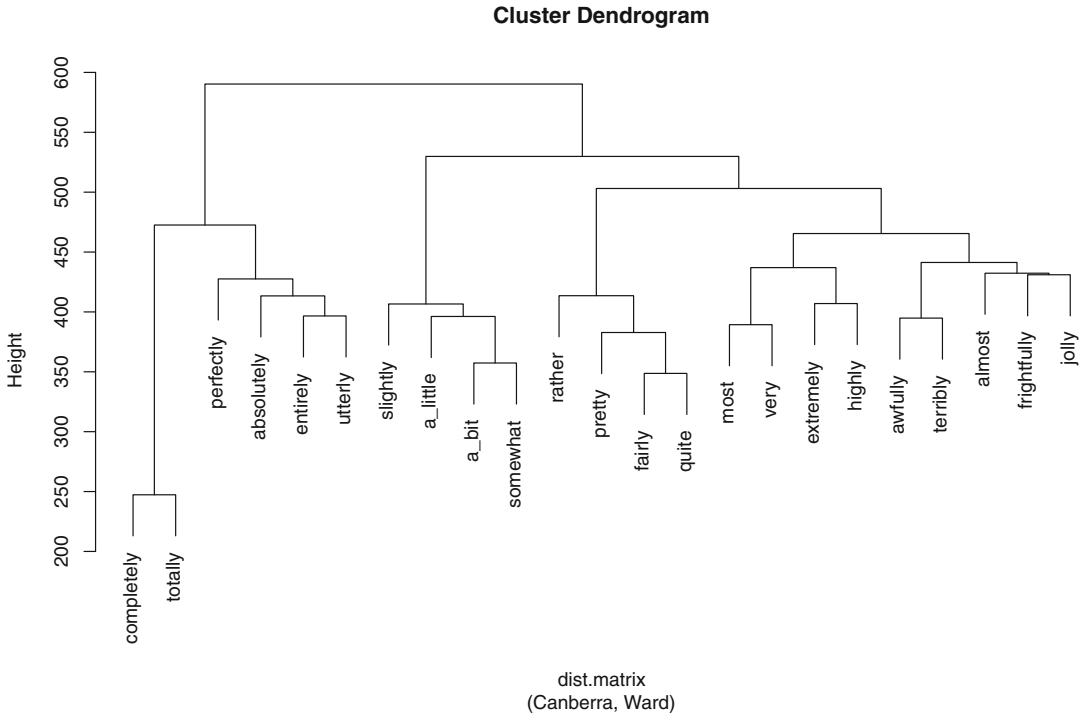


Fig. 10.17: Cluster dendrogram of 23 English intensifiers, clustered according to their adjectival collexemes (distance method: Canberra; amalgamation rule: Ward)

```
> cutree(clusters, 6)
      a_bit    a_little  absolutely      almost     awfully  completely    entirely   extremely
          1           1           2           3           3           4           2           5
     fairly frightfully      highly       jolly        most   perfectly      pretty       quite
          6           3           5           3           5           2           6           6
     rather    slightly    somewhat    terribly     totally     utterly        very
          6           1           1           3           4           2           5
```

The dendrogram displays several homogeneous clusters. Based on the classes delimited by the red rectangles, we find, from left to right: maximizers[15] (*completely*, *totally*, *perfectly*, *absolutely*, *entirely*, and *utterly*), diminishers (*slightly*, *a little*, *a bit*, and *somewhat*), moderators (*rather*, *pretty*, *fairly*, and *quite*), and boosters (*most*, *very*, *extremely*, *highly*, *awfully*, *terribly*, *frightfully*, and *jolly*). The presence of *almost* is odd, but probably due to the fact that it is used as a sentential adverb more than it is used as an adjectival intensifier. In any case, based on the simplicity of the input, HCA reveals consistent clusters.

[15] I use the terminology proposed by Paradis (1997).

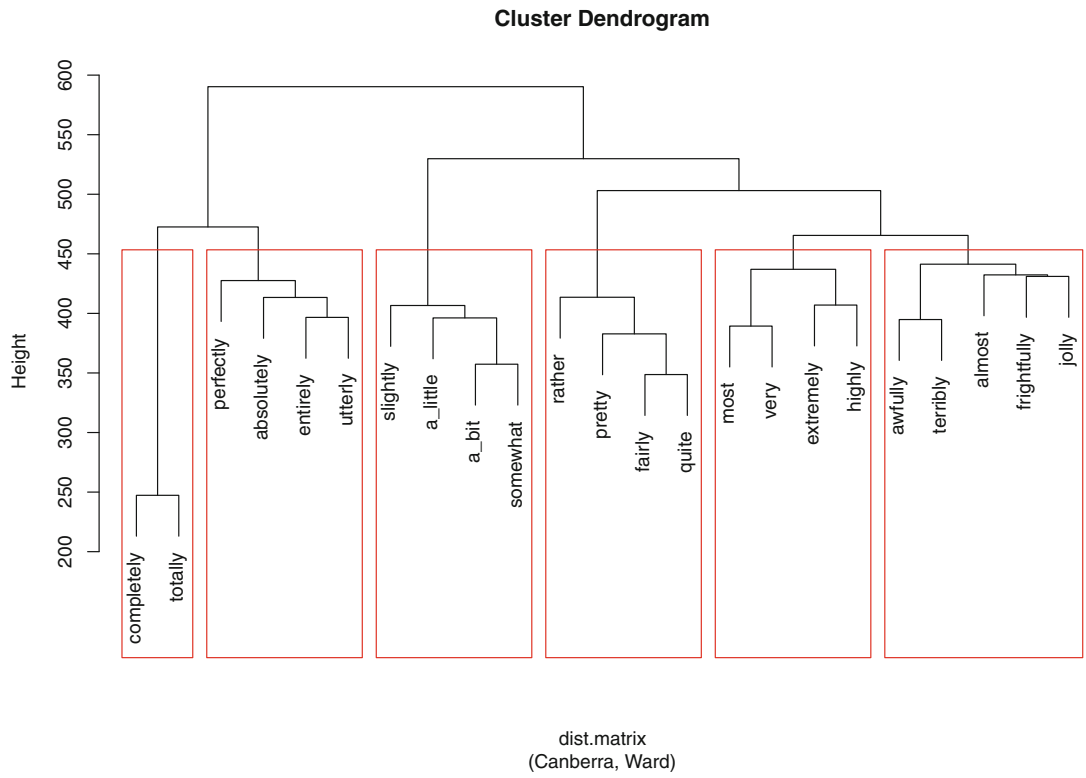


Fig. 10.18: Cluster dendrogram of 23 English intensifiers with 6 groups

10.6.4 Standardizing Variables

When your variables have different units, you must standardize them with the `scale()` function. Fox and Jacewicz (2009) compare the spectral change of five vowels (/ɪ/, /ɛ/, /e/, /æ/, and /aɪ/) in Western North Carolina, Central Ohio, and Southern Wisconsin. The corpus consists of 1920 utterances by 48 female informants. Fox and Jacewicz find variation in formant dynamics as a function of phonetic factors. They also find that, for each vowel and for each measure employed, dialect is a strong source of variation in vowel-inherent spectral change. The data set `hca_vow_cons.rds` is adapted from Tabs. 7 and 8 from Fox and Jacewicz (2009).

```
> rm(list=ls(all=T))
> data <- readRDS("C:/CLSR/chap10/hca_vow_cons.rds") # Windows
> data <- readRDS("/CLSR/chap10/hca_vow_cons.rds") # Mac
```

The data frame consists of 30 observations and 4 variables (type `head(data)` to see the detail).[16] Each observation is named after:

- the vowel of interest (/ɪ/, /ɛ/, /e/, /æ/, or /aɪ/);

[16] The phonetic characters will display properly if IPA fonts are installed on your system.

- the State where the informant is from: Ohio (`Ohio`), North Carolina (`NorthC`), and Wisconsin (`Wisc`);
- the phonetic factor that conditions variation: vowel emphasis (`V_emph`) and consonental context (`C_context`).

For example, `ɪ_NorthC_V_emph` stands for the vowel /ɪ/ in the context of vowel emphasis in the dialect of North Carolina. The four variables come in two units:

- `vow_dur_voiced_ms`: the mean duration of vowels before voiced consonants (in milliseconds),
- `vow_dur_voiceless_ms`: the mean duration of vowels before voiceless consonants (in milliseconds),
- `TL_voiced_Hz`: the mean trajectory length of vowels before voiced consonants (in hertz),
- `TL_voiceless_Hz`: the mean trajectory length of vowels before voiceless consonants (in hertz).

You need to scale the variables as you convert the data table into a distance object.

```
> dist.object <- dist(scale(data), method="manhattan", diag=T, upper=T)
```

Next, amalgamate the clusters and generate the plot (Fig. 10.19).

```
> clusters <- hclust(dist.object, method="ward.D")
> plot(clusters, main="cluster dendrogram", sub="(manhattan, Ward)", cex=0.6)
> rect.hclust(clust.ana, 4) # 4 cluster classes
```

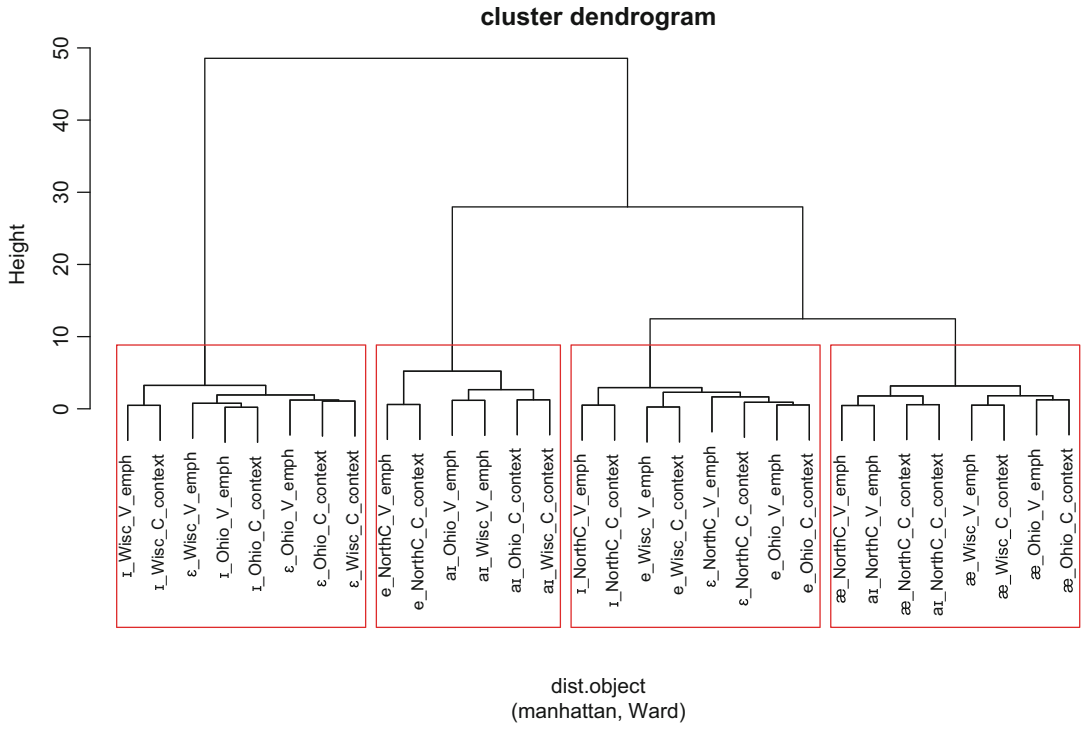


Fig. 10.19: Cluster dendrogram of five vowels in American English

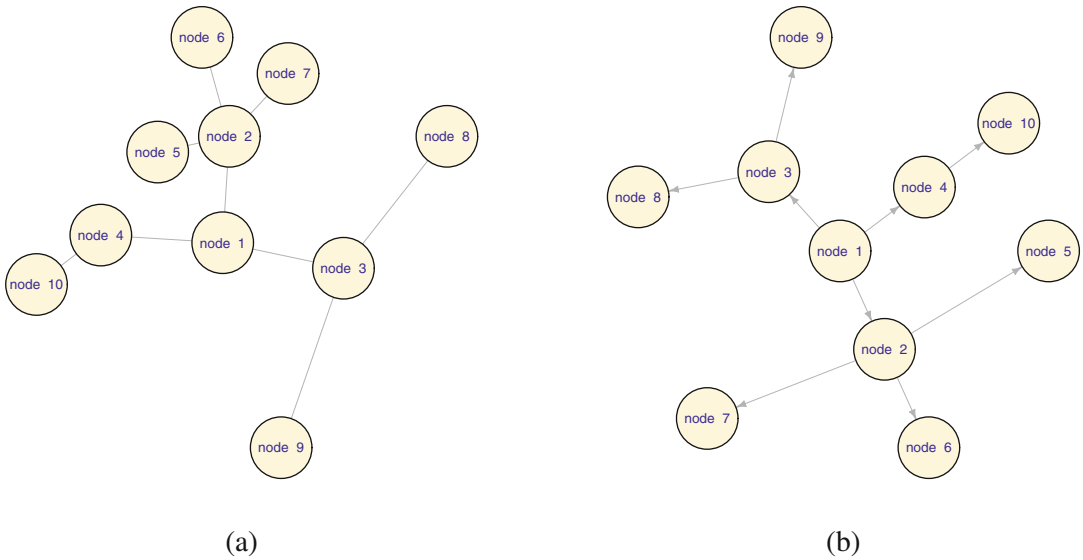
Some homogeneous clusters appear on the dendrogram. The leftmost cluster brings together /ɪ/ and /ɛ/ in Wisconsin and Ohio. The center left cluster groups together /e/ and /aɪ/ in all three states. The center right cluster groups together /ɪ/, /e/, and /ɛ/ in all three states. The rightmost cluster groups together /æ/ and /aɪ/ in all three states.

10.7 Networks

Network science has its roots in graph theory, which originated when Euler solver the Königsberg bridge problem in the eighteenth century.

10.7.1 What Is a Graph?

A graph consists of vertices (nodes) and edges (links). In Fig. 10.20a, each circle is a node and each line is an edge. Each edge denotes a relationship between two nodes. The relationship is undirected, i.e. symmetric. In Sect. 9.3.7, we saw that collocations could be asymmetric. Likewise, the edges of a graph may be asymmetric, i.e. have a direction association with them. The graph in Fig. 10.20b illustrates this asymmetry. In the parlance of graph theory, this graph is directed.



(a) (b)

Fig. 10.20: Two simple graphs. (**a**) An undirected graph; (**b**) a directed graph

There are several R packages that allow you to plot such graphs: `network` (Butts 2008, 2015), `igraph` (Csárdi and Nepusz 2006). My preference goes to `igraph` because it is flexible, and thoroughly documented (Kolaczyk and Csárdi 2014; Arnold and Tilton 2015).

The simplest way of plotting a graph is to do it manually. Let me start with the graph in Fig. 10.20a. First, clear up R's memory, install the `igraph` package, and load it.

```
> rm(list=ls(all=TRUE))
> install.packages("igraph")
> library(igraph)
```

Next, create an igraph object. We plot ten nodes, numbered from 1 to 10. We link the nodes with edges by means of two hyphens (`-`). We use the function `graph.formula()`.

```
> g <- graph.formula(1--2, 1--3, 1--4, 2--5, 2--6, 2--7, 3--8, 3--9, 4--10)
```

The function outputs a graph object.

```
> g
IGRAPH UN-- 10 9 --
+ attr: name (v/c)
+ edges (vertex names):
[1] 1--2  1--3  1--4  2--5  2--6  2--7  3--8  3--9  4--10
```

Given the simplicity of this first graph, this output is far from exciting. It says that we have a graph that is undirected (U) and named (N) with ten nodes and nine edges. We now assign labels to the node.

```
> labels <- c("node 1", "node 2", "node 3", "node 4", "node 5", "node 6", "node 7", "node 8",
+             "node 9", "node 10")
```

`igraph` allows you to specify plotting parameters such as the size, color, and labels of nodes (`vertex.size`, `vertex.color`, `vertex.labels`), the font size (`vertex.label.cex`) and the font family of node labels (`vertex.label.family`), and the width of edges (`edge.width`).

```
> igraph.options(vertex.size=30, vertex.color="cornsilk", vertex.label=labels, vertex.label.cex=0.8,
+                vertex.label.family="Helvetica", edge.width=0.5)
```

The final step is to plot the graph. If you intend to plot the graph several times without altering its shape, use `set.seed()`. The first argument of `plot()` is the graph object. The second argument is the layout of the graph. There are several layouts to choose from: Fruchterman-Reingold (`layout.fruchterman.reingold`), Kamada-Kawai (`layout.kamada.kawai`), Sugiyama (`layout.sugiyama`), Reingold-Tilford (`layout.reingold.tilford`), etc. This choice is important as a graph that appears cluttered with a given layout may appear fine with another layout. For this graph, I use the Reingold-Tilford layout. If you omit `circular=TRUE` inside the `layout.reingold.tilford()` function, the graph will be shaped as a tree.

```
> set.seed(42)
> plot(g, layout=layout.reingold.tilford(g, circular=T))
```

The directed graph is plotted likewise. The only difference is how you link the nodes. The source node and the target node are on either side of `-+`.

```
> g.d <- graph.formula(1-+2, 1-+3, 1-+4, 2-+5, 2-+6, 2-+7, 3-+8, 3-+9, 4-+10)
```

The rest of the code is identical, except for the specification of the arrow size in `igraph.options`: `edge.arrow.size=0.5`.

```
> labels=c("node  1", "node  2", "node  3", "node  4", "node  5", "node  6", "node  7", "node  8",
+          "node  9", "node  10")
> igraph.options(vertex.size=30, vertex.color="cornsilk", vertex.label=labels, vertex.label.cex=0.8,
+                vertex.label.family="Helvetica", edge.width=0.5, edge.arrow.size=0.5)
> plot(g.d, layout=layout.reingold.tilford(g.d, circular=TRUE))
```

10.7.2 The Linguistic Relevance of Graphs

10.7.2.1 An Undirected Graph

The attributes of a graph may be assigned linguistically relevant features. For example, the frequency of a constituent has a correlate in the importance of the node: frequent nodes may be represented with a larger size than infrequent nodes. The co-occurrence frequency of at least two nodes has a correlate in the number of edges between nodes: frequent co-occurrence can be visualized by means of either multiple edges or one edge whose thickness is indexed on frequency.

Our first example is based on the `dative` dataset, from the `languageR` package. We want to plot a graph of the relations between the double-object dative (NP), the prepositional dative (PP), and the verbs that occur in each construction.

First, load and inspect two data sets: `v.attr.dat` and `e.list.dat`.

```
> v.attr.dat <- readRDS("/CLSR/chap10/v.attr.dat.rds") # Windows
> v.attr.dat <- readRDS("/CLSR/chap10/v.attr.dat.rds") # Mac
>
> e.list.dat <- readRDS("C:/CLSR/chap10/elist.dat.rds") # Windows
> e.list.dat <- readRDS("/CLSR/chap10/elist.dat.rds") # Mac
```

`v.attr.dat` contains the node attributes.

```
> head(v.attr.dat)
      Name Frequency
1   accord         1
2   afford         1
3 allocate         3
4    allot         3
5    allow        13
6   assess         1
```

We have two attributes: the name of each node (all the verbs, `NP`, and `PP`) and its frequency. The names will be used to label the nodes, and the frequencies to adjust the size of the nodes accordingly. `e.list.dat` is an edge list.

```
> head(e.list.dat)
  RealizationOfRecipient  Verb
1                     NP  feed
2                     NP  give
5                     NP offer
7                     NP   pay
8                     NP bring
9                     NP teach
```

Each row stands for an edge in the graph. Each co-occurrence between a construction and a verb type is signalled by an edge. The first column contains the origins of the edges and the second column the end points.

One important aspect of graphs is their centrality. Graph centrality is a measure of how important a node is in the context of the entire graph. It can be applied to detect the most prototypical collostructions. There are three popular measures of centrality: degree centrality (nodes are ranked according to the number of edges to which they are connected), eigenvector centrality (nodes connected to important nodes are assigned a higher weight), and betweenness centrality (nodes are ranked according to how many pairs of nodes linked by the shortest path they are connected to). We shall use eigenvector centrality to spot the most influential nodes.

First, we create a graph object with `graph.data.frame()`. We provide the edge list, the node attributes, and specify that we want an undirected graph.

```
> set.seed(33) # for a reproducible graph
> G <- graph.data.frame(e.list.dat, vertices=v.attr.dat, directed="FALSE")
```

Eigenvector centrality is calculated with the `evcent()` function, which outputs a list. We are interested in the `vector` element of the list. It is a vector that assigns each node a numerical score between 0 and 1.

```
> eigenCent <- evcent(G)$vector
> eigenCent
   accord    afford  allocate     allot     allow    assess    assign    assure     award  bequeath
0.1031757 0.1081254 0.1081254 0.2113011 0.1031757 0.1031757 0.2113011 0.1031757 0.2113011 0.1081254
      bet     bring     carry     cause      cede    charge      cost      deal   deliver      deny
0.1031757 0.2113011 0.1081254 0.2113011 0.1081254 0.2113011 0.1031757 0.1081254 0.2113011 0.2113011
       do    extend      feed      fine      flip     float    funnel       get      give     grant
0.2113011 0.2113011 0.2113011 0.1031757 0.1031757 0.1031757 0.1081254 0.1081254 0.2113011 0.2113011
guarantee      hand hand_over     issue     lease     leave      lend      loan      mail      make
0.1031757 0.2113011 0.1081254 0.1081254 0.1081254 0.2113011 0.2113011 0.2113011 0.2113011 0.2113011
      net     offer       owe       pay  pay_back    permit    prepay   present   promise     quote
0.1031757 0.2113011 0.2113011 0.2113011 0.1031757 0.1031757 0.1081254 0.1081254 0.2113011 0.1031757
     read    refuse reimburse     repay    resell       run      sell sell_back  sell_off      send
0.2113011 0.1031757 0.1031757 0.1081254 0.1081254 0.1081254 0.2113011 0.1081254 0.1081254 0.2113011
    serve      show      slip    submit    supply      swap      take     teach      tell    tender
0.2113011 0.2113011 0.1081254 0.1081254 0.1081254 0.1031757 0.2113011 0.2113011 0.2113011 0.1081254
    trade      vote      will      wish     write        NP        PP
0.1081254 0.1031757 0.1031757 0.1031757 0.2113011 0.9542230 1.0000000
```

We are going to use the centrality score to color the nodes. This score is known to drop dramatically[17] We must therefore rearrange the centrality scores into quantized buckets known as "bins". This is how you do it.

```
> bins <- unique(quantile(eigenCent, seq(0,1,length.out=30)))
> vals <- cut(eigenCent, bins, labels=FALSE, include.lowest=TRUE)
```

Each bin is now assigned a color along a rainbow continuum skewed to the reds (for higher scores) and yellows (for lower scores). The color values are assigned to the color attributes of the nodes. This means that the plotting function will shade the nodes according to their eigenvector centralities.

```
> colorVals <- rev(heat.colors(length(bins)))[vals]
> V(G)$color <- colorVals
```

Next, we adjust the size of the nodes based on `Frequency` and we assign a label to each name (`Name`).

```
> v.size <- 0.7*sqrt(V(G)$Frequency)+.5 # adjust node size
> v.label <- V(G)$Name
```

[17] Most of the nodes have a score that is well below 0.5. See it for yourself by entering `plot(sort(eigenCent, decreasing=TRUE), type="l")`.

Finally, we select a layout (Fruchterman-Reingold) and plot the graph with the desired parameters. Note that, to make the labels more legible, a distance is set between each label and the center of its node (`vertex.label.dist=0.2`). Additionally, label size is further adjusted because it is not set along the same scale as node size.

```
> l <- layout.fruchterman.reingold(G)
> plot(G, layout=l, vertex.size=v.size, vertex.label=v.label, vertex.label.dist=0.2,
+      vertex.label.cex=0.3*sqrt(v.size), vertex.label.family="Arial")
```

The resulting graph is displayed in Fig. 10.21. Among other things, the graph allows you to see that:

- the double-object dative (NP) is more frequent than the prepositional dative in the corpus;
- three classes of verbs emerge: the verbs that co-occur exclusively with the double-object dative, the verbs that co-occur exclusively with the prepositional dative, and the verbs that co-occur with both alternants;
- *give*, whose meaning is the most consonant with the meaning of the dative construction, occurs with the highest frequency and is compatible with both alternants.

10.7.2.2 A Directed Graph

Our second example is based on the study of *A as NP* (Desagulier 2016). We want to plot a graph of asymmetric collostructions between the adjective slot and the NP slot. Load the following data sets: `v.attr.anp` and `e.list.anp`.

```
> v.attr.anp <- readRDS("/CLSR/chap10/v.attr.anp.rds") # Windows
> v.attr.anp <- readRDS("/CLSR/chap10/v.attr.anp.rds") # Mac
>
> e.list.anp <- readRDS("C:/CLSR/chap10/elist.anp.rds") # Windows
> e.list.anp <- readRDS("/CLSR/chap10/elist.anp.rds") # Mac
```

`v.attr.anp` contains the node attributes: `Name` (i.e. the adjective types), `Type` (`A` or `NP`) and `Const_freq` (the construction's frequency). The names will be used to label the nodes, the types to distinguish adjectives from NPs based on color,[18] and the construction's frequency to adjust the size of the nodes.

```
> head(v.attr.anp)
         Name Type Const_freq
1         air   NP          4
2       angel   NP          2
3       angry    A          3
4  anthracite   NP          2
5       arrow   NP          2
6 baby_s_skin   NP          2
```

`e.list.anp` contains the list of edges. Like above, the first column is the origin of the edge, and the second column the end point. Based on ΔP, a constituent (adjective or NP) that attracts the constituent in the other slot is placed in the second column. If not, it is placed in the first column. Therefore, each column contains adjectives and NPs. The third column is an edge attribute, namely the collostruction strength of the association between the adjective and the NP measured with G^2. We are going to use this edge attribute to adjust the width of each edge.

[18] This time, we do not use a centrality measure.

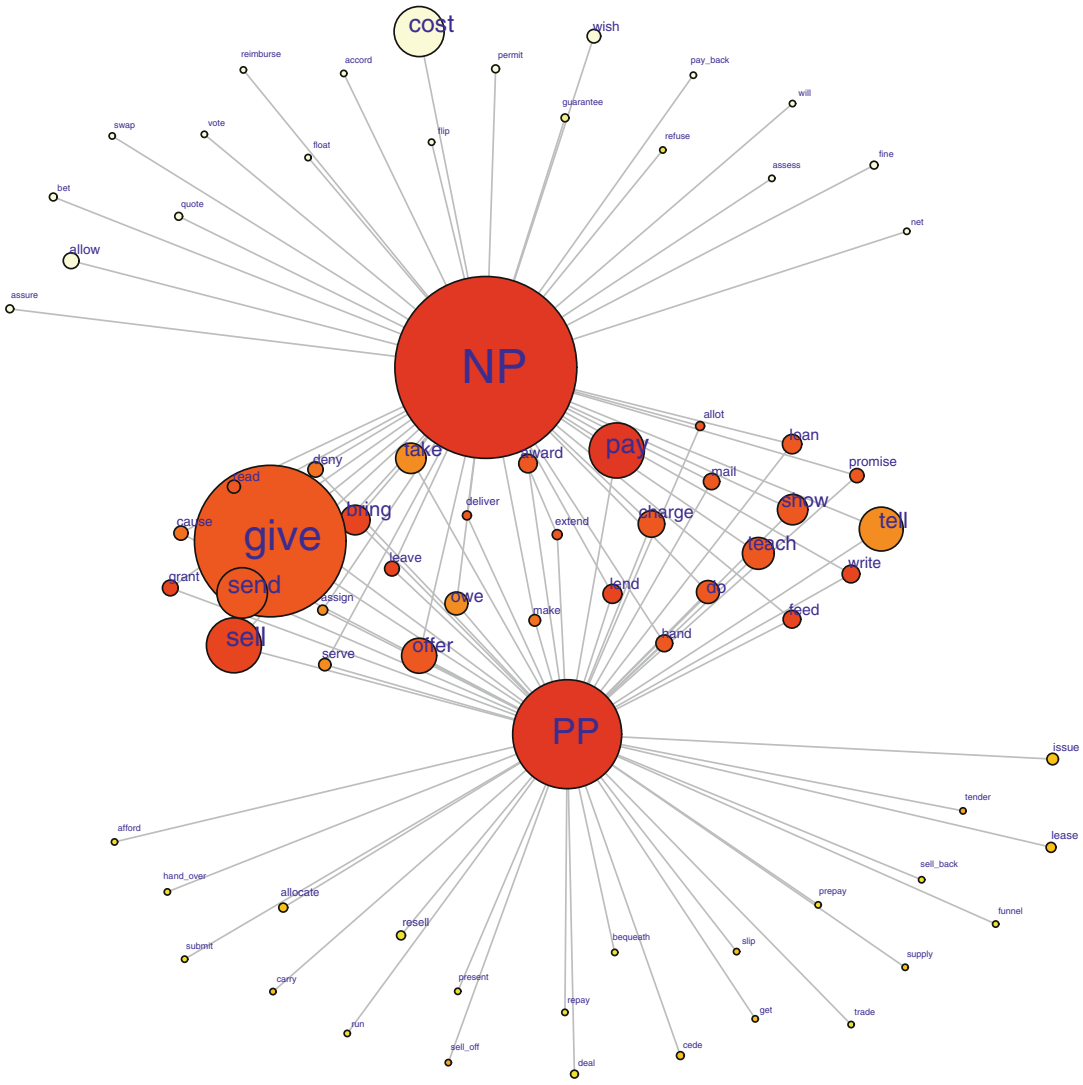


Fig. 10.21: A graph of the verbs recruited by two English dative alternants (NP and PP)

```
> head(e.list.anp)
                from    to coll_strength
1               man   old       7.0591
2             chalk white      11.4292
3            marble white      11.4292
4 night_follows_day  sure      11.5171
5 money_in_the_bank  good      12.8812
6            spring clear      13.5610
```

First, we create a graph object with the same function as above: `graph.data.frame()`. This time, we specify `directed="TRUE"`.

```
> set.seed(15) # for a reproducible graph
> G <- graph.data.frame(e.list.anp, vertices=v.attr.anp, directed="TRUE")
> G
IGRAPH DN-- 231 211 --
+ attr: name (v/c), Type (v/c), Const_freq (v/n), coll_strength (e/n)
+ edges (vertex names):
 [1] man              ->old    chalk            ->white  marble           ->white
 [4] night_follows_day->sure   money_in_the_bank->good   spring           ->clear
 [7] anthracite       ->black  world            ->old    paper            ->white
[10] baby_s_skin      ->smooth monkey           ->quick  syrup            ->smooth
[13] mud              ->clear  midnight         ->black  muck             ->soft
[16] putty            ->soft   shite            ->soft   granite          ->hard
[19] cream            ->smooth crystal          ->clear  coal             ->black
[22] thunder          ->black  mice             ->quiet  shit             ->thick
+ ... omitted several edges
```

Upon inspection, we see that the graph is indeed directed and named. We also have the detail of the attributes of the nodes and edges. Next, we adjust the size of the nodes based on construction frequency and assign a label to each node.

```
> v.size <- 1.5*sqrt(V(G)$Const_freq)+.5
> v.label <- V(G)$name
```

We weight the edges based on collostruction strength. We are going to use this weighting to adjust the width of the edges when we plot the graph.

```
> E(G)$weight <- E(G)$coll_strength
```

We want two different shapes and two different colors for the nodes depending on whether they denote adjectives or NPs. We convert `Type` into a numeric vector and assign shapes and colors accordingly.

```
> type <- as.factor(V(G)$Type)
> type <- as.numeric(type)
> v.shapes <- c("circle","square")[type]
> v.colors <- c("red", "dodgerblue")[type]
```

Finally, we select a layout (Fruchterman-Reingold again) and plot the graph. As planned, we use `E(G)$weight` to adjust the width of the edges and the size of the arrows (because the graph is directed, the edges are automatically converted into arrows).

```
> l <- layout.fruchterman.reingold(G)
> plot(G, layout=l, edge.width=2.5*sqrt(E(G)$weight), edge.arrow.size=1.5*sqrt(E(G)$weight),
+      vertex.size=v.size, vertex.label=v.label, vertex.shape=v.shapes, vertex.color= v.colors,
+      vertex.label.cex=v.size, vertex.label.color="black", vertex.label.family="Arial")
```

The resulting graph is displayed in Fig. 10.22. Among other things, the graph shows that some types are somewhat fixed as per the adjectives or NPs that they collocate with (*large as life*, *honest as the day is long*, *gauche as a schoolgirl*, *thin as a rake*, *taut as a bowstring*, etc.) On the other hand, some other types are more productive. They are part of more complex combinatorial constellations of adjectives and NPs (towards the center part of the graph). These complex networks are based on hubs, i.e. constituents that are connected to several other nodes, e.g. the adjectives *white*, *cold*, *clear* or *smooth* or the NP *hell*. We can also see that some attractive constituents are themselves attracted by other constituents. For example, the adjective *sure* attracts the NPs *death* and *night follows day*. At the same time, it is attracted by the NP *hell*.

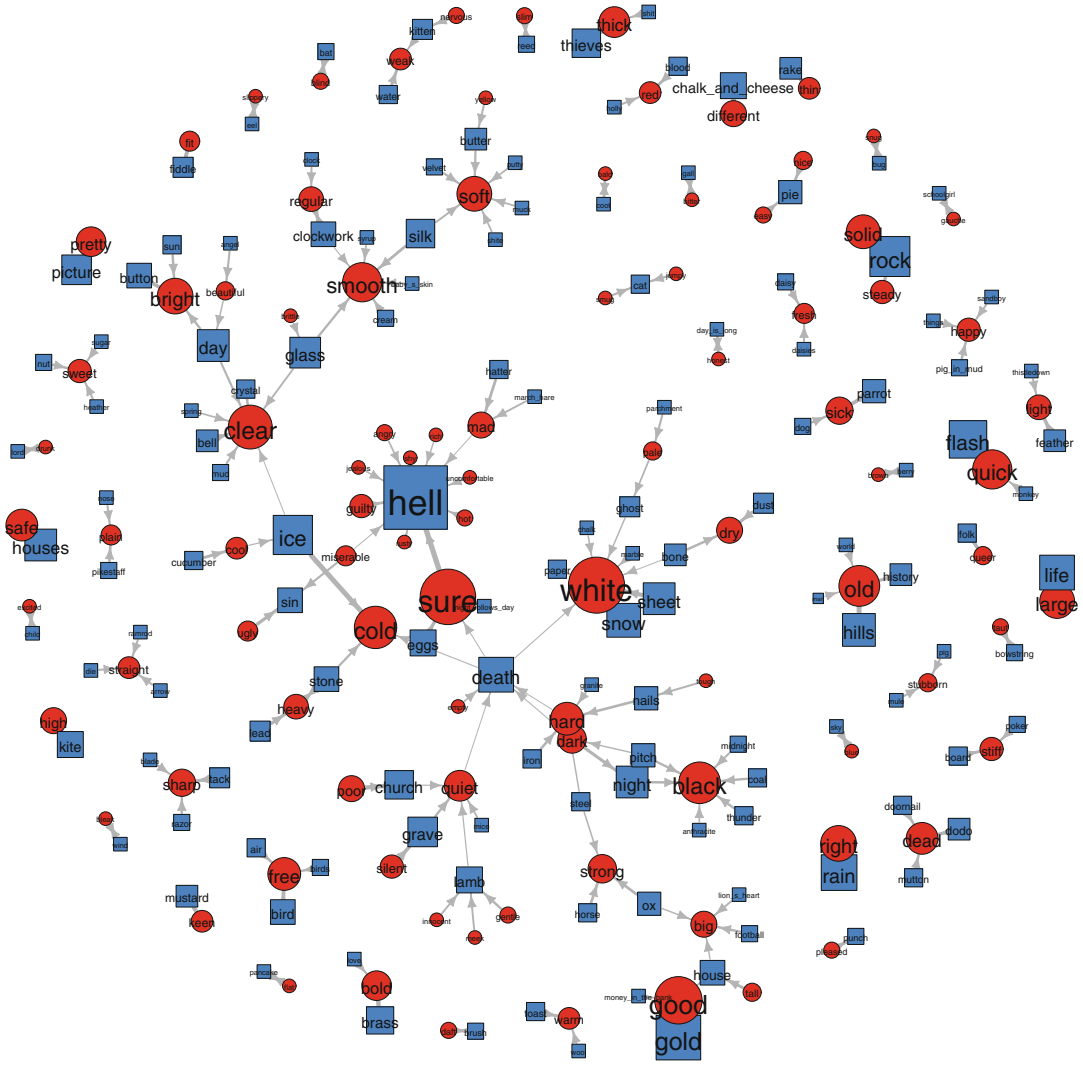


Fig. 10.22: A graph of asymmetric collostructions between adjectives and NPs in *A as NP* (adjectives are in *red*, NPs in *blue*)

Exercises

10.1. Principal component analysis

The data set for this exercise is `it.be.A.to.V.that.rds`. You will find it in `(C:)/CLSR/chap10`. It is based on a study of the productivity of the construction *it BE ADJ to V that* in the British National Corpus (see Exercise 5.2).

The data frame consists of 244 observations and 6 variables. The observations are the constituents of the *it BE ADJ to V that* constructions, i.e. either adjectives or verbs in the infinitive. The variables are the following:

- `category`: the grammatical category of the constituent (`ADJ` for adjectives, and `V_inf` for verbs in the infinitive);
- `num.hapax`: the number of times each constituent appears as a hapax legomena in the construction;
- `V`: the type frequency of each constituent in the construction;
- `delta.diff`: the mean difference $\Delta P_{(V\_inf|ADJ)} - \Delta P_{(ADJ|V\_inf)}$ (see Sect. 9.3.7.2);
- `association.chisq`: the mean association score between ADJ and V_inf according to χ^2 (see Sect. 9.3.3.3);
- `P.potential`: the mean $\mathscr{P}$ score of each constituent (see Sect. 9.4.1);
- `P.global`: the mean P^* score of each constituent (see Sect. 9.4.1).[19]

Run a PCA of `it.be.A.to.V.that.rds`. Consider the variables `num.hapax` and `V` supplementary. Decide how many components (or dimensions) should be inspected, plot the results, and interpret them.

10.2. Correspondence analysis

The data set `modals.BNC.rds` is a contingency table that contains the frequencies of ten English modals across seventy text genres in the British National Corpus.[20] Run a CA of `modals.BNC.rds`, which you will find in `(C:)/CLSR/chap10`.

First, plot only the variables that correspond to written texts (the variable names starts with `W_`). What can you say about the respective distributions of *may* and *might*? Second, plot only the variables that correspond to spoken texts (the variable names starts with `S_`). What profile stands out the most? What is the contextual specificity of *will*?

10.3. Multiple correspondence analysis

The data set for this exercise is `typocorp.rds`. You will find it in `(C:)/CLSR/chap10`. It corresponds to Tab. 3.1. The individuals and the variables are described in Sect. 3.1. Your goal is to run a MCA using the `MCA()` function from the `FactoMineR` package. The variables `corpus`, `stored_data_format`, and `size` should be considered supplementary. Plot the graph of individuals. You should obtain Fig. 3.1.

10.4. Hierarchical cluster analysis

Using the same data frame as in Exercise 10.2, run a hierarchical cluster analysis of the ten English modals and plot the results in the form of a dendrogram. The `hclust()` function from the `stats` package in base R should do the trick. Compare the following distance measures:

- Euclidean,
- Manhattan,
- Minkowski,
- Canberra.[21]

Use Ward's method to amalgamate the clusters.

[19] Sometimes, the value of P^* is `Inf` (infinite). Because R cannot make any computation based on `Inf`, the value was changed to 999999999.

[20] See David Lee's Genre Classification Scheme: `http://rdues.bcu.ac.uk/bncweb/manual/genres.html`.

[21] Enter `?dist` for more information about the distance metrics.

10.5. Networks

In `/CLSR/chap10`, you will find two files to plot a network graph of the constituents of *it BE ADJ to V_inf that* (see Ex. 10.1): `v.attr.practice.rds` and `e.list.practice.rds`. The first file contains the vertex attributes. The vertices are the adjectives and the infinitive verbs. There are two attributes: the grammatical category (`A` for adjective and `V` for verb), and the construction frequency. The second file is an edge list. The first column is the origin of the edge, and the second column the end point. The origins and the end points were determined on the basis of the asymmetric association measure ΔP. The third column is an edge attribute. This attribute is the measure of symmetric association between the adjective and the verb according to χ^2. Plot a directed graph of the constituents of the construction. Select only those constituents whose construction frequency is greater than or equal to 20. Choose the `lgl` layout.[22]

References

Abdi, Hervé, and Lynne J. Williams. 2010. Principal Component Analysis. *Wiley Interdisciplinary Reviews: Computational Statistics* 2 (4): 433–459. doi:10.1002/wics.101.

Arnold, Taylor, and Lauren Tilton. 2015. *Humanities Data in R: Exploring Networks, Geospatial Data, Images, and Text*. Quantitative Methods in the Humanities and Social Sciences. New York: Springer.

Baayen, R. Harald. 2008. *Analyzing Linguistic Data: A Practical Introduction to Statistics Using R*. Cambridge: Cambridge University Press.

Benzécri, Jean-Paul. 1973. *L'analyse des données*. L'analyse des correspondances, Vol. 2. Paris, Bruxelles, Montréal: Dunod.

Benzécri, Jean-Paul. 1984. *Analyse des correspondances, exposé élémentaire*. Pratique de l'analyse des données, Vol. 1. Paris: Dunod.

Bourdieu, Pierre. 1979. *La distinction: critique sociale du jugement*. Paris: Les Editions de Minuit.

Butts, Carter T. 2008. `network`: A Package for Managing Relational Data in R. *Journal of Statistical Software* 24 (2). `http://www.jstatsoft.org/v24/i02/paper`

Butts, Carter T. 2015. *`network`: Classes for Relational Data*. R Package Version 1.13.0. `http://CRAN.R-project.org/package=network`.

Camiz, Sergio, and Gastão Coelho Gomes. 2013. Joint Correspondence Analysis Versus Multiple Correspondence Analysis: A Solution to an Undetected Problem. In Classification and Data Mining, ed. Giusti, Antonio, Ritter, Gunter, and Vichi, Maurizio, 11–18. Berlin/Heidelberg: Springer. doi:`10.1007/978-3-642-28894-4_2`.

Cheshire, Jenny. 2007. Discourse Variation, Grammaticalisation and Stuff Like That. *Journal of Sociolinguistics* 11 (2), 155–193. ISSN:1467-9841. doi:1.1111/j.1467-9841.2007.00317x. `http://dx.doi.org/10.1111/j.1467-9841.2007.00317.x`.

Csárdi, Gábor, and Tamas Nepusz. 2006. The igraph Software Package for Complex Network Research. *InterJournal Complex Systems*: 1695. `http://igraph.org`

Darwin, Charles. 1859. *On the Origin of Species by Means of Natural Selection*. London: J. Murray.

Davies, Mark. 2008–2012. *The Corpus of Contemporary American English: 450 Million Words, 1990–present*. `http://corpus.byu.edu/coca/`.

de Leeuw, Jan, and Patrick Mair. 2009. Simple and Canonical Correspondence Analysis Using the R Package anacor. *Journal of Statistical Software* 31 (5), 1–18. `http://www.jstatsoft.org/v31/i05/`.

[22] After loading the `igraph` package, enter `?layout_with_lgl` for details.

Desagulier, Guillaume. 2014. Visualizing distances in a set of near synonyms: *rather*, *quite*, *fairly*, and *pretty*. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 145–178. Amsterdam: John Benjamins.

Desagulier, Guillaume. 2015. Forms and Meanings of Intensification: A Multifactorial Comparison of *Quite* and *Rather*. *Anglophonia* 20. doi:10.400/anglophonia.558. http://anglophonia.revues.org/558.

Desagulier, Guillaume. 2016. A Lesson from Associative Learning: Asymmetry and Productivity in Multiple-Slot Constructions. *Corpus Linguistics and Linguistic Theory* 12 (1): 173–219

Divjak, Dagmar, and Nick Fieller. 2014. Finding Structure in Linguistic Data. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 405–441. Amsterdam: John Benjamins.

Divjak, Dagmar, and Stefan Thomas Gries. 2006. Ways of Trying in Russian: Clustering Behavioral Profiles. *Corpus Linguistics and Linguistic Theory* 2 (1), 23–60.

Divjak, Dagmar, and Stefan Thomas Gries. 2008. Clusters in the Mind?: Converging Evidence from Near Synonymy in Russian. *The Mental Lexicon* 3 (2), 188–213. ISSN: 18711340. http://dx.doi.org/10.1075/ml.3.2.03div.

Dixon, Robert M.W., and Alexandra Y. Aikhenvald. 2004. *Adjective Classes: A Cross-Linguistic Typology*. Oxford: Oxford University Press.

Everitt, Brian S., et al. 2011. *Cluster Analysis*. Wiley Series in Probability and Statistics, Vol. 5, 330–330. Oxford: Wiley-Blackwell. ISBN: 9780470749913 (hbk.): 1755.00; 0470749911 (hbk.): 1755.00.

Fox, Robert Allen, and Ewa Jacewicz. 2009. Cross-Dialectal Variation in Formant Dynamics of American English Vowels. *The Journal of the Acoustical Society of America* 126 (5): 2603–2618. doi:10.1121/1.3212921.

Glynn, Dylan. 2014. Correspondence Analysis: Exploring Data and Identifying Patterns. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*. Human Cognitive Processing, ed. Dylan Glynn and Justyna A. Robinson, 443–485. Amsterdam: John Benjamins.

Greenacre, Michael J. 2007. *Correspondence Analysis in Practice*. Interdisciplinary Statistics Series, Vol. 2. Boca Raton: Chapman Hall/CRC.

Greenacre, Michael J., and Jorg Blasius. 2006. *Multiple Correspondence Analysis and Related Methods*. Boca Raton: Chapman Hall/CRC.

Gries, Stefan Thomas. 2010. *Statistics for Linguistics with R: A Practical Introduction*. Berlin, New York: MoutonDe Gruyter. ISBN: 9783110205657.

Gries, Stefan Thomas, and Anatol Stefanowitsch. 2010. Cluster Analysis and the Identification of Collexeme Classes. In *Empirical and Experimental Methods in Cognitive/Functional Research*, ed. John Newman and Sally Rice, 73–90. Stanford, CA: CSLI.

Husson, François, Sebastien Lê, and Jérôme Pagès. 2011. *Exploratory Multivariate Analysis by Example Using R*. Chapman Hall/CRC Computer Science and Data Analysis. Boca Raton: Chapman Hall/CRC Press.

Husson, François, Julie Josse, Sébastien Lê, and Jéméry Mazet. 2015 *FactoMineR: Multivariate Exploratory Data Analysis and Data Mining*. R Package Version 1.31.4. http://CRAN.R-project.org/package=FactoMineR.

Kolaczyk, Eric D., and Gábor Csárdi. 2014. *Statistical Analysis of Network Data with R*. Use R!, xii, 386. New York, London: Springer.

Lacheret, Anne, Fleury, Serge, Beliaõ, Julie, and Desagulier, Guillaume. Statistical Analyses of Prosodic Structures in Spoken French. In *Rhapsodie. A Prosodic-Syntactic Treebank for Spoken French,* ed. A. Lacheret, S. Kahane, and P. Pietrandrea. Amsterdam: John Benjamins (to appear a)

Lacheret, Anne, Kahane, Sylvain, and Pietrandrea, Paola. *Rhapsodie. A Prosodic-Syntactic Treebank for Spoken French*. Amsterdam: John Benjamins (to appear b)

Le Roux, Brigitte. 2010. *Multiple Correspondence Analysis*. London: SAGE.

Levin, Beth. 1993. *English Verb Classes and Alternations: A Preliminary Investigation*. Chicago, London: University of Chicago Press.

Mikolov, Tomas, et al. 2013a. Distributed Representations of Words and Phrases and Their Compositionality. CoRR abs/1310.4546. http://arxiv.org/abs/1310.4546.

Mikolov, Tomas, et al. 2013b. Efficient Estimation of Word Representations in Vector Space. CoRR abs/1301.3781. http://arxiv.org/abs/1301.3781.

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig. 2013c. Linguistic Regularities in Continuous Space Word Representations. In *Proceedings of NAACL-HLT*, 746–751. http://www.aclweb.org/anthology/N/N13/N131090.pdf.

Nenadic, Oleg, and Michael Greenacre. 2007. Correspondence Analysis in R, with Two- and Three-Dimensional Graphics: The ca Package. *Journal of Statistical Software* 20 (3): 1–13. http://www.jstatsoft.org.

Paradis, Carita. 1997. Degree Modifiers of Adjectives in Spoken British English. Lund: Lund University Press.

Pearson, Karl. 1901. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine* 2 (11): 559–572. doi:10.1080/14786440109462720.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 1532–1543. http://www.aclweb.org/anthology/D14-1162.

The British National Corpus. 2007. *BNC XML Edition*. Version 3. http://www.nat.corp.ox.ac.uk/. Distributed by Oxford University Computing Services on behalf of the BNC Consortium.

van der Maaten, Laurens, and Geoffrey Hinton. 2008. Visualizing Data Using t-SNE. *Journal of Machine Learning Research* 9: 2579–2605.

Ward, Joe H. 1963. Hierarchical Grouping to Optimize an Objective Function. *Journal of the American Statistical Association* 58 (301): 236. ISSN:01621459. http://dx.doi.org/10.2307/2282967.

Appendix A

A.1 Chapter 6

A.1.1 Dispersion Plots

```
> ############################
> # CORPUS FILE PREPARATION #
> ############################
>
> rm(list=ls(all=TRUE))
> corpus.file <- scan("/CLSR/BNC_baby/A1E.xml", what="character", sep="\n", fileEncoding="UTF-8")
> head(corpus.file)
> corpus.file.sentences <- grep("<s n=", corpus.file, perl=TRUE, value=TRUE)
> head(corpus.file.sentences)
>
> library(gsubfn)
>
> #######
> # NN0 #
> #######
>
> tagged.NN0 <- unlist(strapply(corpus.file.sentences,
+                               "<w c5=\"NN0(-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                               perl=T, backref=1))
> head(tagged.NN0)
>
> NN0<-gsub("<.*?>", "", tagged.NN0, perl=T)
> NN0<-gsub("(^ | +$)", "", NN0, perl=T)
> head(NN0)
>
> untagged.words<-unlist(strsplit(corpus.file.sentences, "<.*?>", perl=TRUE))
> head(untagged.words)
> untagged.words <- gsub("[,;:.?!'`-]", "", untagged.words)
> untagged.words <- gsub("(^ +| +$)", "", untagged.words, perl=TRUE)
> untagged.words <- untagged.words[nchar(untagged.words)>0]
> head(untagged.words)
>
> seq.corpus <- seq(1:length(untagged.words))
>
> NN0.positions <- integer()
> for (i in 1:length(NN0)) {
+  NN0.position <- which(untagged.words==NN0[i])
+  NN0.positions <- c(NN0.positions, NN0.position)
+ }
```


G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8

```
> head(NN0.positions)
>
> NN0.count <- rep(NA, length(seq.corpus))
>
> NN0.count[NN0.positions] <- 1
>
> #######
> # NN1 #
> #######
>
> tagged.NN1 <- unlist(strapply(corpus.file.sentences,
+                               "<w c5=\"NN1(-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                               perl=T, backref=1))
> head(tagged.NN1)
>
> NN1<-gsub("<.*?>", "", tagged.NN1, perl=T)
> NN1<-gsub("(^ | +$)", "", NN1, perl=T)
> head(NN1)
>
> NN1.positions <- integer()
> for (i in 1:length(NN1)) {
+  NN1.position <- which(untagged.words==NN1[i])
+  NN1.positions <- c(NN1.positions, NN1.position)
+ }
> head(NN1.positions)
>
> NN1.count <- rep(NA, length(seq.corpus))
>
> NN1.count[NN1.positions] <- 1
>
> #######
> # NN2 #
> #######
>
> tagged.NN2 <- unlist(strapply(corpus.file.sentences,
+                               "<w c5=\"NN2(-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                               perl=T, backref=1))
> head(tagged.NN2)
>
> NN2<-gsub("<.*?>", "", tagged.NN2, perl=T)
> NN2<-gsub("(^ | +$)", "", NN2, perl=T)
> head(NN2)
>
> NN2.positions <- integer()
> for (i in 1:length(NN2)) {
+  NN2.position <- which(untagged.words==NN2[i])
+  NN2.positions <- c(NN2.positions, NN2.position)
+ }
> head(NN2.positions)
>
> NN2.count <- rep(NA, length(seq.corpus))
>
> NN2.count[NN2.positions] <- 1
>
> #######
> # NP0 #
> #######
>
> tagged.NP0 <- unlist(strapply(corpus.file.sentences,
+                               "<w c5=\"NP0(-...)?\" hw=\"\\w+\" pos=\"SUBST\">\\w+ ?</w>",
+                               perl=T, backref=1))
> head(tagged.NP0)
>
> NP0<-gsub("<.*?>", "", tagged.NP0, perl=T)
> NP0<-gsub("(^ | +$)", "", NP0, perl=T)
> head(NP0)
>
> NP0.positions <- integer()
```

```
> for (i in 1:length(NP0)) {
+  NP0.position <- which(untagged.words==NP0[i])
+  NP0.positions <- c(NP0.positions, NP0.position)
+ }
> head(NP0.positions)
>
> NP0.count <- rep(NA, length(seq.corpus))
>
> NP0.count[NP0.positions] <- 1
>
> #########
> # PLOTS #
> #########
>
> par(mfrow=c(4,1))
> plot(NN0.count, main="", xlab="A1E.xml", ylab="NN0", type="h", ylim=c(0,1), yaxt='n')
> plot(NN1.count, main="", xlab="A1E.xml", ylab="NN1", type="h", ylim=c(0,1), yaxt='n')
> plot(NN2.count, main="", xlab="A1E.xml", ylab="NN2", type="h", ylim=c(0,1), yaxt='n')
> plot(NP0.count, main="", xlab="A1E.xml", ylab="NP0", type="h", ylim=c(0,1), yaxt='n')
```

A.2 Chapter 8

A.2.1 Contingency Table

```
> # a frequency list of tags in A1J.xml of BNC-Baby
> rm(list=ls(all=TRUE))
> library(gsubfn)
> corpus.file <- scan("/CLSR/BNC_baby/A1J.xml", what="char", sep="\n")
> sentences <- grep("<s n=", corpus.file, value=TRUE)
> tags <- unlist(strapplyc(sentences, "w c5=\"(.*?)\""))
> table_tags <- sort(table(tags), decreasing=TRUE)
> freqlist_tags <- paste(names(table_tags), table_tags, sep="\t")
> head(freqlist_tags)
> cat("TAG\tFREQUENCY", freqlist_tags, file="~/Desktop/freqlist_tags_A1J.txt", sep="\n")
> freqlist.A1J <- read.table("~/Desktop/freqlist_tags_A1J.txt", header=TRUE)
> saveRDS(freqlist.A1J, "/CLSR/chap8/freqlist.A1J.rds")
> freqlist.A1J <- readRDS("/CLSR/chap8/freqlist.A1J.rds")
> str(freqlist.A1J)
> nouns <- c("NN0", "NN1", "NN2", "NP0")
> freqlist.nouns <- freqlist.A1J[freqlist.A1J$TAG %in% nouns, ]
> sum(freqlist.nouns$FREQUENCY)
>
> # a frequency list of tags in A1K.xml of BNC-Baby
> corpus.file <- scan("/CLSR/BNC_baby/A1K.xml", what="char", sep="\n")
> sentences <- grep("<s n=", corpus.file, value=TRUE)
> tags <- unlist(strapplyc(sentences, "w c5=\"(.*?)\""))
> table_tags <- sort(table(tags), decreasing=TRUE)
> freqlist_tags <- paste(names(table_tags), table_tags, sep="\t")
> cat("TAG\tFREQUENCY", freqlist_tags, file="~/Desktop/freqlist_tags_A1K.txt", sep="\n")
> freqlist.A1K <- read.table("~/Desktop/freqlist_tags_A1K.txt", header=TRUE)
> saveRDS(freqlist.A1K, "/CLSR/chap8/freqlist.A1K.rds")
> freqlistA1K <- readRDS("/CLSR/chap8/freqlist.A1K.rds")
>
> # a frequency list of tags in A1L.xml of BNC-Baby
> corpus.file <- scan("/CLSR/BNC_baby/A1L.xml", what="char", sep="\n")
> sentences <- grep("<s n=", corpus.file, value=TRUE)
> tags <- unlist(strapplyc(sentences, "w c5=\"(.*?)\""))
> table_tags <- sort(table(tags), decreasing=TRUE)
> freqlist_tags <- paste(names(table_tags), table_tags, sep="\t")
> cat("TAG\tFREQUENCY", freqlist_tags, file="~/Desktop/freqlist_tags_A1L.txt", sep="\n")
```

```
> freqlist.A1L <- read.table("~/Desktop/freqlist_tags_A1L.txt", header=TRUE)
> saveRDS(freqlist.A1L, "/CLSR/chap8/freqlist.A1L.rds")
> freqlist.A1L <- readRDS("/CLSR/chap8/freqlist.A1L.rds")
>
> freqlist.A1J <- readRDS("/CLSR/chap8/freqlist.A1J.rds")
> freqlist.A1K <- readRDS("/CLSR/chap8/freqlist.A1K.rds")
> freqlist.A1L <- readRDS("/CLSR/chap8/freqlist.A1L.rds")
>
> freqlist.NN0.A1J <- freqlist.A1J[freqlist.A1J$TAG %in% "NN0", ]
> freqlist.NN1.A1J <- freqlist.A1J[freqlist.A1J$TAG %in% "NN1", ]
> freqlist.NN2.A1J <- freqlist.A1J[freqlist.A1J$TAG %in% "NN2", ]
> freqlist.NP0.A1J <- freqlist.A1J[freqlist.A1J$TAG %in% "NP0", ]
>
> freqlist.NN0.A1K <- freqlist.A1K[freqlist.A1K$TAG %in% "NN0", ]
> freqlist.NN1.A1K <- freqlist.A1K[freqlist.A1K$TAG %in% "NN1", ]
> freqlist.NN2.A1K <- freqlist.A1K[freqlist.A1K$TAG %in% "NN2", ]
> freqlist.NP0.A1K <- freqlist.A1K[freqlist.A1K$TAG %in% "NP0", ]
>
> freqlist.NN0.A1L <- freqlist.A1L[freqlist.A1L$TAG %in% "NN0", ]
> freqlist.NN1.A1L <- freqlist.A1L[freqlist.A1L$TAG %in% "NN1", ]
> freqlist.NN2.A1L <- freqlist.A1L[freqlist.A1L$TAG %in% "NN2", ]
> freqlist.NP0.A1L <- freqlist.A1L[freqlist.A1L$TAG %in% "NP0", ]
>
> NN0.A1J <- sum(freqlist.NN0.A1J$FREQUENCY)
> NN1.A1J <- sum(freqlist.NN1.A1J$FREQUENCY)
> NN2.A1J <- sum(freqlist.NN2.A1J$FREQUENCY)
> NP0.A1J <- sum(freqlist.NP0.A1J$FREQUENCY)
>
> NN0.A1K <- sum(freqlist.NN0.A1K$FREQUENCY)
> NN1.A1K <- sum(freqlist.NN1.A1K$FREQUENCY)
> NN2.A1K <- sum(freqlist.NN2.A1K$FREQUENCY)
> NP0.A1K <- sum(freqlist.NP0.A1K$FREQUENCY)
>
> NN0.A1L <- sum(freqlist.NN0.A1L$FREQUENCY)
> NN1.A1L <- sum(freqlist.NN1.A1L$FREQUENCY)
> NN2.A1L <- sum(freqlist.NN2.A1L$FREQUENCY)
> NP0.A1L <- sum(freqlist.NP0.A1L$FREQUENCY)
>
> cont.table <- matrix(c(NN0.A1J, NN1.A1J, NN2.A1J, NP0.A1J,
+                        NN0.A1K, NN1.A1K, NN2.A1K, NP0.A1K, NN0.A1L,
+                        NN1.A1L, NN2.A1L, NP0.A1L), nrow=4, ncol=3, byrow=F); cont.table
> row.names(cont.table) <- c("NN0", "NN1", "NN2", "NP0")
> colnames(cont.table) <- c("A1J.xml", "A1K.xml", "A1L.xml")
> install.packages("epitools")
> library(epitools)
> table.margins(cont.table)
```

A.2.2 Discrete Probability Distributions

```
> par(mfrow=c(3,2))
>
> n=4
> prob=0.5
> x=0:4
> p=dbinom(x, n, prob)
> plot(x, p, type="h", xlim=c(-1,5), ylim=c(0,.6),
+      lwd=2, col="blue", ylab="p", xlab="number of vowels", main="4 trials")
> #points(x,p,pch=16,cex=2,col="black")
> text(0, p[1]+.1, p[1])
> text(1, p[2]+.1, p[2])
> text(2, p[3]+.1, p[3])
> text(3, p[4]+.1, p[4])
```

```
> text(4, p[5]+.1, p[5], col="red")
>
> n=10
> prob=0.5
> x=0:10
> p=dbinom(x, n, prob)
> plot(x, p, type="h", xlim=c(-1,11), ylim=c(0,.4),
+      lwd=2, col="blue", ylab="p", xlab="number of vowels", main="10 trials")
> #points(x,p,pch=16,cex=2,col="black")
> text(10, p[11]+.1, p[11], col="red")
> Arrows(10,0.09,10,p[11]+.02, col="red")
>
>
> n=20
> prob=0.5
> x=0:20
> p=dbinom(x, n, prob)
> plot(x, p, type="h", xlim=c(-1,21), ylim=c(0,.3),
+      lwd=2, col="blue", ylab="p", xlab="number of vowels", main="20 trials")
> #points(x,p,pch=16,cex=2,col="black")
> text(19, p[21]+.1, p[21], col="red", cex=0.8)
> Arrows(20,0.09,20,p[21]+.02, col="red")
>
> n=30
> prob=0.5
> x=0:30
> p=dbinom(x, n, prob)
> plot(x, p, type="h", xlim=c(-1,31), ylim=c(0,.3), lwd=2,
+      col="blue", ylab="p", xlab="number of vowels", main="30 trials")
> #points(x,p,pch=16,cex=2,col="black")
> text(28, p[31]+.1, p[31], col="red", cex=0.8)
> Arrows(30,0.09,30,p[31]+.02, col="red")
>
> n=50
> prob=0.5
> x=0:50
> p=dbinom(x, n, prob)
> plot(x, p, type="h", xlim=c(-1,51), ylim=c(0,.2),
+      lwd=2, col="blue", ylab="p", xlab="number of vowels", main="50 trials")
> #points(x,p,pch=16,cex=2,col="black")
> text(45, p[51]+.1, p[51], col="red", cex=0.8)
> Arrows(50,0.09,50,p[51]+.009, col="red")
>
> n=100
> prob=0.5
> x=0:100
> p=dbinom(x, n, prob)
> plot(x, p, type="h", xlim=c(-1,101), ylim=c(0,.2),
+      lwd=2, col="blue", ylab="p", xlab="number of vowels", main="100 trials")
> #points(x,p,pch=16,cex=2,col="black")
> text(90, p[101]+.1, p[101], col="red", cex=0.8)
> Arrows(100,0.09,100,p[101]+.009, col="red")
```

A.2.3 A χ^2 Distribution Table

Table A.1

degrees of freedom	Probability p of obtaining a larger value of χ^2					
	0.1	0.05	0.025	0.01	0.005	0.001
1	2.7055	3.8415	5.0239	6.6349	7.8794	10.8276
2	4.6052	5.9915	7.3778	9.2103	10.5966	13.8155
3	6.2514	7.8147	9.3484	11.3449	12.8382	16.2662
4	7.7794	9.4877	11.1433	13.2767	14.8603	18.4668
5	9.2364	11.0705	12.8325	15.0863	16.7496	20.515
6	10.6446	12.5916	14.4494	16.8119	18.5476	22.4577
7	12.017	14.0671	16.0128	18.4753	20.2777	24.3219
8	13.3616	15.5073	17.5345	20.0902	21.955	26.1245
9	14.6837	16.919	19.0228	21.666	23.5893	27.8772
10	15.9872	18.307	20.4832	23.2093	25.1882	29.5883
11	17.275	19.6751	21.92	24.725	26.7568	31.2641
12	18.5493	21.0261	23.3367	26.217	28.2995	32.9095
13	19.8119	22.362	24.7356	27.6882	29.8195	34.5282
14	21.0641	23.6848	26.1189	29.1412	31.3193	36.1233
15	22.3071	24.9958	27.4884	30.5779	32.8013	37.6973
16	23.5418	26.2962	28.8454	31.9999	34.2672	39.2524
17	24.769	27.5871	30.191	33.4087	35.7185	40.7902
18	25.9894	28.8693	31.5264	34.8053	37.1564	42.3124
19	27.2036	30.1435	32.8523	36.1909	38.5823	43.8202
20	28.412	31.4104	34.1696	37.5662	39.9968	45.3147
21	29.6151	32.6706	35.4789	38.9322	41.4011	46.797
22	30.8133	33.9244	36.7807	40.2894	42.7957	48.2679
23	32.0069	35.1725	38.0756	41.6384	44.1813	49.7282
24	33.1962	36.415	39.3641	42.9798	45.5585	51.1786
25	34.3816	37.6525	40.6465	44.3141	46.9279	52.6197
26	35.5632	38.8851	41.9232	45.6417	48.2899	54.052
27	36.7412	40.1133	43.1945	46.9629	49.6449	55.476
28	37.9159	41.3371	44.4608	48.2782	50.9934	56.8923
29	39.0875	42.557	45.7223	49.5879	52.3356	58.3012
30	40.256	43.773	46.9792	50.8922	53.672	59.7031
40	51.8051	55.7585	59.3417	63.6907	66.766	73.402
50	63.1671	67.5048	71.4202	76.1539	79.49	86.6608
60	74.397	79.0819	83.2977	88.3794	91.9517	99.6072
70	85.527	90.5312	95.0232	100.4252	104.2149	112.3169
80	96.5782	101.8795	106.6286	112.3288	116.3211	124.8392
90	107.565	113.1453	118.1359	124.1163	128.2989	137.2084
100	118.498	124.3421	129.5612	135.8067	140.1695	149.4493

Appendix B
Bibliography

Abdi, Hervé, and Lynne J. Williams. 2010. Principal Component Analysis. *Wiley Interdisciplinary Reviews: Computational Statistics* 2 (4): 433–459. doi:10.1002/wics.101.

Agresti, Alan. 2002. *Categorical Data Analysis*. Wiley Series in Probability and Statistics, 2nd ed. New York: Wiley-Interscience.

Albert, Jim, and Maria Rizzo. 2012. *R by Example*. Use R! New York: Springer.

Allan, Lorraine G. 1980. A Note on Measurement of Contingency Between Two Binary Variables in Judgment Tasks. *Bulletin of the Psychonomic Society* 15 (3): 147–149.

Andor, József. 2004. The Master and His Performance: An Interview with Noam Chomsky. *Intercultural Pragmatics* 1 (1), 93–111. doi:10.1515/iprg.2004.009.

Arnold, Taylor, and Lauren Tilton. 2015. *Humanities Data in R: Exploring Networks, Geospatial Data, Images, and Text*. Quantitative Methods in the Humanities and Social Sciences. New York: Springer.

Baayen, R. Harald. 2008. *Analyzing Linguistic Data: A Practical Introduction to Statistics Using R*. Cambridge: Cambridge University Press.

Baayen, Rolf Harald. 1989. *A Corpus-Based Approach to Morphological Productivity. Statistical Analysis and Psycholinguistic Interpretation*. Amsterdam: Centrum Wiskunde en Informatica.

Baayen, Rolf Harald. 1993. On Frequency, Transparency and Productivity. In *Yearbook of Morphology 1992*, ed. Geert Booij and Jaap van Marle, 181–208. Dordrecht, London: Kluwer.

Baayen, Rolf Harald. 2001. *Word Frequency Distributions*. Dordrecht: Kluwer Academic Publishers.

Baayen, Rolf Harald. 2009. Corpus Linguistics in Morphology: Morphological Productivity. In *Corpus Linguistics. An International Handbook*, ed. Anke Lüdeling and Merja Kytö, 899–919. Berlin: Mouton de Gruyter.

Baayen, Rolf Harald. 2011. Corpus Linguistics and Naive Discriminative Learning. *Brazilian Journal of Applied Linguistics* 11: 295–328.

Baayen, Rolf Harald, and Rochelle Lieber. 1991. Productivity and English Derivation: A Corpus-Based Study. *Linguistics* 29: 801–843.

Baroni, Marco, et al. 2009. The WaCky Wide Web: A Collection of Very Large Linguistically Processed Web-Crawled Corpora. *Language Resources and Evaluation* 43 (3): 209–226.

Bauer, Laurie. 2001. *Morphological Productivity*. Cambridge: Cambridge University Press.

Benzécri, Jean-Paul. 1973. *L'analyse des données*. L'analyse des correspondances, Vol. 2. Paris, Bruxelles, Montréal: Dunod.

Benzécri, Jean-Paul. 1984. *Analyse des correspondances, exposé élémentaire*. Pratique de l'analyse des données, Vol. 1. Paris: Dunod.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8

Biber, Douglas. 1993. Representativeness in Corpus Design. *Literary and Linguistic Computing* 8 (4): 241–257.

Biber, Douglas, Susan Conrad, and Randi Reppen. 1998. *Corpus Linguistics: Investigating Language Structure and Use*. Cambridge: Cambridge University Press.

Boas, Hans C., and Sarah Schuchard. 2012. A Corpus-Based Analysis of Preterite Usage in Texas German. In *Proceedings of the 34th Annual Meeting of the Berkeley Linguistics Society*.

Bourdieu, Pierre. 1979. *La distinction: critique sociale du jugement*. Paris: Les Editions de Minuit.

Burns, Patrick. 2011. *The R Inferno*. http://www.burns-stat.com/pages/Tutor/R_inferno.pdf (visited on 08/26/2015).

Butts, Carter T. 2008. Network: A Package for Managing Relational Data in R. *Journal of Statistical Software* 24 (2). http://www.jstatsoft.org/v24/i02/paper.

Butts, Carter T. 2015. *Network: Classes for Relational Data*. R Package Version 1.13.0. http://CRAN.R-project.org/package=network.

Bybee, Joan. 2010. *Language Usage and Cognition*. Cambridge: Cambridge University Press.

Bybee, Joan L. 2006. From Usage to Grammar: The Mind's Response to Repetition. *Language* 82 (4): 711–733.

Camiz, Sergio, and Gastão Coelho Gomes. 2013. Joint Correspondence Analysis Versus Multiple Correspondence Analysis: A Solution to an Undetected Problem. In Classification and Data Mining, ed. Antonio Giusti, Gunter Ritter, and Maurizio Vichi, 11–18. Berlin, Heidelberg: Springer. doi:10.1007/9783642288944_2.

Cheshire, Jenny. 2007. Discourse Variation, Grammaticalisation and Stuff Like That. *Journal of Sociolinguistics* 11 (2): 155–193. ISSN: 1467-9841. doi:1.1111/j.1467-9841.2007.00317x. http://dx.doi.org/10.1111/j.1467-9841.2007.00317.x.

Chomsky, Noam. 1957. *Syntactic Structures*. The Hague: Mouton.

Chomsky, Noam. 1962. *Aspects of the Theory of Syntax*. Cambridge, MA: MIT Press.

Chomsky, Noam. 1995. *The Minimalist Program*. Cambridge, MA: MIT Press.

Church, Kenneth, and Patrick Hanks. 1990. Word Association Norms, Mutual Information, and Lexicography. *Computational Linguistics* 16 (1): 22–29.

Church, Kenneth, et al. 1991. Using Statistics in Lexical Analysis. In *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, ed. Uri Zernik, 115–164. Hillsdale: Lawrence Erlbaum.

Cohen, Ayala. 1980. On the Graphical Display of the Significant Components in a Two-Way Contingency Table. *Communications in Statistics - Theory and Methods* 9 (10): 1025–1041.

Crawley, Michael. 2012. *The R Book*, 2nd ed. Chichester: Wiley.

Cressie, Noel and Timothy R.C. Read. 1989. Pearson's χ^2 and the Loglikelihood Ratio Statistic G^2: A Comparative Review. *International Statistical Review* 57 (1), 19–43. ISSN: 03067734, 17515823. http://www.jstor.org/stable/1403582.

Crowdy, Steve. 1993. Spoken Corpus Design. *Literary and Linguistic Computing* 8 (4): 259–265.

Csárdi, Gábor, and Tamas Nepusz. 2006. The igraph Software Package for Complex Network Research. *InterJournal Complex Systems*: 1695. http://igraph.org.

Dalgaard, Peter. 2008. *Introductory Statistics with R*, 2nd ed. New York: Springer.

Darwin, Charles. 1859. *On the Origin of Species by Means of Natural Selection*. London: J. Murray.

Davies, Mark. 1859. *Corpus of Canadian English: 50 Million Words, 1920s–2000s*. http://corpus.byu.edu/can/.

Davies, Mark. 2008–2012. *The Corpus of Contemporary American English: 450 Million Words, 1990–Present*. http://corpus.byu.edu/coca/.

Davies, Mark. 2010. *The Corpus of Historical American English: 400 Million Words, 1810–2009*. http://corpus.byu.edu/coca/.

de Leeuw, Jan, and Patrick Mair. 2009. Simple and Canonical Correspondence Analysis Using the R Package anacor. *Journal of Statistical Software* 31 (5): 1–18. http://www.jstatsoft.org/v31/i05/.

Desagulier, Guillaume. 2014. Visualizing Distances in a Set of Near Synonyms: *Rather*, *Quite*, *Fairly*, and *Pretty*. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 145–178. Amsterdam: John Benjamins.

Desagulier, Guillaume. 2015. Forms and Meanings of Intensification: A Multifactorial Comparison of *Quite* and *Rather*. *Anglophonia* 20. doi:10.400/anglophonia558. http://anglophonia.revues.org/558.

Desagulier, Guillaume. 2016. A Lesson from Associative Learning: Asymmetry and Productivity in Multiple-Slot Constructions. *Corpus Linguistics and Linguistic Theory* 12 (1): 173–219.

Dik, Simon C. 1978. *Functional Grammar*. Amsterdam, Oxford: North-Holland Publishing Co.

Dik, Simon C. 1997. *The Theory of Functional Grammar*, ed. Kees Hengeveld, 2nd ed. Berlin, New York: Mouton de Gruyter.

Divjak, Dagmar, and Nick Fieller. 2014. Finding Structure in Linguistic Data. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 405–441. Amsterdam: John Benjamins.

Divjak, Dagmar, and Stefan Thomas Gries. 2006. Ways of Trying in Russian: Clustering Behavioral Profiles. *Corpus Linguistics and Linguistic Theory* 2 (1): 23–60.

Divjak, Dagmar, and Stefan Thomas Gries. 2008. Clusters in the Mind?: Converging Evidence from Near Synonymy in Russian. *The Mental Lexicon* 3 (2): 188–213. ISSN: 18711340. http://dx.doi.org/10.1075/ml.3.2.03div.

Dixon, Robert M.W., and Alexandra Y. Aikhenvald. 2004. *Adjective Classes: A Cross-Linguistic Typology*. Oxford: Oxford University Press.

Dunning, Ted. 1993. Accurate Methods for the Statistics of Surprise and Coincidence. *Computational Linguistics* 19 (1): 61–74.

Ellis, Nick. 2006. Language Acquisition as Rational Contingency Learning. *Applied Linguistics* 27 (1), 1–24.

Ellis, Nick, and Fernando Ferreira Junior. 2009. Constructions and Their Acquisition: Islands and the Distinctiveness of Their Occupancy. *Annual Review of Cognitive Linguistics* 7: 187–220.

Everitt, Brian S., et al. 2011. *Cluster Analysis*. Wiley Series in Probability and Statistics, Vol. 5, 330. Oxford: Wiley-Blackwell. ISBN: 9780470749913 (hbk.): 1755.00; 0470749911 (hbk.): 1755.00.

Evert, Stefan. 2005. The Statistics of Word Cooccurrences: Word Pairs and Collocations. PhD Dissertation. Universität Stuttgart. http://elib.uni-stuttgart.de/opus/volltexte/2005/2371/pdf/Evert2005phd.pdf (Visited on 08/14/2015).

Evert, Stefan. 2009. Corpora and Collocations. In *Corpus Linguistics: An International Handbook*, ed. Anke Lüdeling and Merja Kytö, Vol. 2, 1212–1248. Berlin, New York: Mouton de Gruyter.

Evert, Stefan, and Marco Baroni. 2006. The `zipfR` Library: Words and Other Rare Events in R. Presentation at useR! 2006: The Second R User Conference, Vienna, Austria.

Evert, Stefan, and Marco Baroni. 2007. `zipfR`: Word Frequency Distributions in R. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics on Interactive Posters and Demonstration Sessions*, 29–32. R Package Version 0.6–6 of 2012-04-03), Prague, Czech Republic.

Ferraresi, Adriano. 2007. Building a Very Large Corpus of English Obtained by Web Crawling: ukWaC. Master's Thesis. University of Bologna.

Firth, John. 1957. A Synopsis of Linguistic Theory 1930–1955. In *Selected Papers of J.R. Firth 1952–1959*, ed. Frank Palmer, 168–205. London: Longman.

Fisher, Ronald Aylmer Sir. 1935. *The Design of Experiments*. Edinburgh, London: Oliver Boyd.

Fox, Robert Allen, and Ewa Jacewicz. 2009. Cross-Dialectal Variation in Formant Dynamics of American English Vowels. *The Journal of the Acoustical Society of America* 126 (5): 2603–2618. doi:10.1121/1.3212921.

Francis, W. Nelson, and Henry Kučera. 1979. *Manual of Information to Accompany a Standard Corpus of Present-Day Edited American English, for Use with Digital Computers*. Department of Linguistics. Brown University. http://www.hit.uib.no/icame/brown/bcm.html (visited on 03/10/2015).

Friendly, Michael. 2000. *Visualizing Categorical Data*. Cary, NC: SAS Publications.

Garside, Roger. 1987. The CLAWS Word-Tagging System. In *The Computational Analysis of English: A Corpus-Based Approach*, ed. Roger Garside, Geoffrey Leech, and Geoffrey Sampson, 30–41. London: Longman.

Geeraerts, Dirk. 2010. The Doctor and the Semantician. In *Quantitative Methods in Cognitive Semantics: Corpus-Driven Approaches*, ed. Dylan Glynn and Kerstin Fischer, 63–78. Berlin, Boston: Mouton De Gruyter.

Gesmann, Markus, and Diego de Castillo. 2011. Using the Google Visualisation API with R. *The R Journal* 3 (2): 40–44. https://journal.r-project.org/archive/2011-2/RJournal_20112_Gesmann+de~Castillo.pdf.

Ghadessy, Mohsen, Alex Henry, and Robert L. Roseberry. 2001. *Small Corpus Studies and ELT*. Amsterdam: John Benjamins.

Givón, Talmy. 1995. *Functionalism and Grammar*. Amsterdam: John Benjamins.

Glynn, Dylan. 2010. Corpus-Driven Cognitive Semantics: Introduction to the Field. In *Quantitative Methods in Cognitive Semantics: Corpus-Driven Approaches*, 1–42. Berlin: Mouton de Gruyter.

Glynn, Dylan. 2014. Correspondence Analysis: Exploring Data and Identifying Patterns. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*. Human Cognitive Processing, ed. Dylan Glynn and Justyna A. Robinson, 443–485. Amsterdam: John Benjamins.

Goldberg, Adele E. 1995. *Constructions: A Construction Grammar Approach to Argument Structure*. Chicago: University of Chicago Press.

Goldberg, Adele E. 2003. Constructions: A New Theoretical Approach to Language. *Trends in Cognitive Sciences* 7 (5): 219–224. http://dx.doi.org/10.1016/S1364-6613(03)00080-9.

Greenacre, Michael J. 2007. *Correspondence Analysis in Practice*. Interdisciplinary Statistics Series, Vol. 2. Boca Raton: Chapman Hall/CRC.

Greenacre, Michael J., and Jorg Blasius. 2006. *Multiple Correspondence Analysis and Related Methods*. Boca Raton: Chapman Hall/CRC.

Greenberg, Joseph H. 1963. Some Universals of Grammar with Particular Reference to the Order of Meaningful Elements. In *Universals of Human Language*, ed. Joseph H. Greenberg, 73–113. Cambridge: MIT Press.

Gries, Stefan Thomas. 2001. A Corpus-Linguistic Analysis of *-ic* and *-ical* Adjectives. *ICAME Journal* 25: 65–108. http://clu.uni.no/icame/ij25/gries.pdf.

Gries, Stefan Thomas. 2009a. *Quantitative Corpus Linguistics with R: A Practical Introduction*. New York, NY: Routledge.

Gries, Stefan Thomas. 2009b. What is Corpus Linguistics? *Language and Linguistics Compass* 3: 1225–1241. doi:10.1111/j.1749818X.2009.00149.x.

Gries, Stefan Thomas. 2010. *Statistics for Linguistics with R: A Practical Introduction*. Berlin, New York: MoutonDe Gruyter. ISBN: 9783110205657.

Gries, Stefan Thomas. 2013. 50-Something Years of Work on Collocations: What is or Should be Next ... *International Journal of Corpus Linguistics* 18 (1), 137–166.

Gries, Stefan Thomas. 2014a. *Coll. Analysis 3.5. A Script for R to Compute Perform Collostructional Analyses*. University of California, Santa Barbara. http://www.linguistics.ucsb.edu/faculty/stgries/teaching/groningen/coll.analysis.r (visited on 07/09/2016).

Gries, Stefan Thomas. 2014b. Frequency Tables: Tests, Effect Sizes, and Explorations. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 365–389. Amsterdam: John Benjamins.

Gries, Stefan Thomas, and Anatol Stefanowitsch. 2004a. Co-Varying Collexemes in the *Into*-causative. In *Language Culture and Mind*, ed. Michel Achard and Suzanne Kemmer, 225–236. Stanford: CSLI.

Gries, Stefan Thomas, and Anatol Stefanowitsch. 2004b. Extending Collostructional Analysis: A Corpus-Based Perspective on 'Alternations'. *International Journal of Corpus Linguistics* 9(1): 97–129.

Gries, Stefan Thomas, and Anatol Stefanowitsch. 2010. Cluster Analysis and the Identification of Collexeme Classes. In *Empirical and Experimental Methods in Cognitive/Functional Research*, ed. John Newman and Sally Rice, 73–90. Stanford, CA: CSLI.

Haspelmath, Martin. 2011. The Indeterminacy of Word Segmentation and the Nature of Morphology and Syntax. *Folia Linguistica* 45 (1): 31–80.

Hilpert, Martin. 2011. Dynamic Visualizations of Language Change: Motion Charts on the Basis of Bivariate and Multivariate Data from Diachronic Corpora. *International Journal of Corpus Linguistics* 16 (4): 435–461. http://dx.doi.org/10.1075/ijcl.16.4.01hil. http://www.jbe-platform.com/content/journals/10.1075/ijcl.16.4.01hil.

Hilpert, Martin. 2014. Collostructional Analysis: Measuring Associations Between Constructions and Lexical Elements. In *Corpus Methods for Semantics: Quantitative Studies in Polysemy and Synonymy*, ed. Dylan Glynn and Justyna Robinson, 391–404. Amsterdam: John Benjamins.

Hofland, Knut, and Stig Johansson. 1982. *Word frequencies in British and American English*. Bergen: Norwegian Computing Centre for the Humanities.

Hollmann, Willem B., and Anna Siewierska. 2007. A Construction Grammar Account of Possessive Constructions in Lancashire Dialect : Some Advantages and Challenges. *English Language and Linguistics* 11 (2): 407–424. doi:10.1017/S1360674307002304.

Hunston, Susan. 2002. *Corpora in Applied Linguistics*. Cambridge: Cambridge University Press.

Husson, Francois, Sébastien Lê, and Jérôme Pagès. 2011. *Exploratory Multivariate Analysis by Example Using R*. Chapman Hall/CRC Computer Science and Data Analysis. Boca Raton: CRC Press.

Husson, François, Julie Josse, Sébastien Lê, and Jéméry Mazet. 2015. *FactoMineR: Multivariate Exploratory Data Analysis and Data Mining*. R Package Version 1.31.4. http://CRAN.Rproject.org/package=FactoMineR.

Isel, Frédéric, Thomas C. Gunter, and Angela D. Friederici. 2003. "Prosody-Assisted Head-Driven Access to Spoken German Compounds. *Journal of Experimental Psychology* 29 (2): 277–288. doi:10.1037/02787393.29.2.277.

Jockers, Matthew. 2014. *Text Analysis with R for Students of Literature*. New York: Springer.

Johansson, Stig, Geoffrey Leech, and Helen Goodluck. 1978. *Manual of Information to Accompany the Lancaster-Oslo/Bergen Corpus of British English, for Use with Digital Computers*. Department of English. University of Oslo. http://clu.uni.no/icame/manuals/LOB/INDEX.HTM (visited on 03/10/2015).

Kennedy, Graeme. 1998. *An Introduction to Corpus Linguistics*. Harlow: Longman.

Kennedy, Graeme. 2003. Amplifier Collocations in the British National Corpus: Implications for English Language Teaching. *TESOL Quarterly* 37 (3): 467–487.

Kilgarriff, Adam. 2001. Comparing Corpora. *International Journal of Corpus Linguistics* 6 (1): 97–133.

Kilgarriff, Adam. 2005. Language is Never Ever Ever Random. *Corpus Linguistics and Linguistic Theory* 1 (2): 263–276. http://dx.doi.org/10.1515/cllt.2005.1.2.263.

Kolaczyk, Eric D., and Gábor Csárdi. 2014. *Statistical Analysis of Network Data with R*. Use R!, xii, 386 New York, London: Springer.

Kotze, Theunis J. van W., and D.V. Gokhale. 1980. A Comparison of the Pearson χ^2 and Log-Likelihood-Ratio Statistics for Small Samples by Means of Probability Ordering. *Journal of Statistical Computation and Simulation* 12 (1), 1–13. doi:10.1080/00949658008810422. eprint: http://dxdoiorg/10.108/00949658008810422.

Labov, William. 1975. Empirical Foundations of Linguistic Theory. In *The Scope of American Linguistics: Papers of the First Golden Anniversary Symposium of the Linguistic Society of America*, ed. Robert Austerlitz, 77–133. Lisse: Peter de Ridder.

Lacheret, Anne, Sylvain Kahane, and Paola Pietrandrea. 2017. *Rhapsodie. A Prosodic-Syntactic Tree-bank for Spoken French*. Amsterdam, Philadelphia: John Benjamins (to appear).

Lafon, Pierre. 1980. Sur la variabilité de la fréquence des formes dans un corpus. *Mots* 1: 127–165.

Lafon, Pierre. 1981. Analyse lexicométrique et recherche des cooccurrences. *Mots*: 95–148. ISSN: 0243-6450. http://wwwperseefr/web/revues/home/prescript/article/mots_0243-6450_1981_num_3_1_1041.

Lafon, Pierre. 1984. *Dépouillements et statistiques en lexicométrie*. Travaux de linguistique quantitative. Genève, Paris: Slatkine, Champion.

Langacker, Ronald W. 1987. *Foundations of Cognitive Grammar: Theoretical Prerequisites*, Vol. 1. Stanford: Stanford University Press.

Langacker, Ronald W. 1988. A Usage-Based Model. In *Topics in Cognitive Linguistics*, ed. Brygida Rudzka-Ostyn, 127–161. Amsterdam, Philadelphia: John Benjamins.

Langacker, Ronald W. 1991. *Foundations of Cognitive Grammar: Descriptive Application*, Vol. 2. Stanford: Stanford University Press.

Langacker, Ronald W. 2008. *Cognitive Grammar : A Basic Introduction*. Oxford: Oxford University Press.

Le Roux, Brigitte. 2010. *Multiple Correspondence Analysis*. London: SAGE.

Lebart, Ludovic, André Salem, and Lisette Berry. 1998. *Exploring Textual Data*. Text, Speech and Language Technology. Dordrecht, Boston, London: Kluwer Academic Publishers.

Leech, Geoffrey. 2007. New Resources, or Just Better Old Ones? The Holy Grail of Representativeness. In *Corpus Linguistics and the Web*, ed. Marianne Hundt, Nadja Nesselhauf, and Carolin Biewer, 133–149. Amsterdam: Rodopi.

Leech, Goeffrey, and Roger Fallon. 1992. Computer Corpora – What do They Tell Us About Culture? In *ICAME*, Vol. 16, 29–50.

Levin, Beth. 1993. *English Verb Classes and Alternations: A Preliminary Investigation*. Chicago, London: University of Chicago Press.

Manning, Christopher D., and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge: MIT Press.

McEnery, Tony, and Andrew Hardie. 2012. *Corpus Linguistics : Method, Theory and Practice*. Cambridge Textbooks in Linguistics. Cambridge, New York: Cambridge University Press.

McEnery, Tony, and Richard Xiao. 2005. Character Encoding in Corpus Construction. In *Developing Linguistic Corpora: A Guide to Good Practice*, ed. Martin Wynne, 47–58. Oxford: Oxbow Books.

Meyer, Charles F. 2002. *English Corpus Linguistics: An Introduction*. Cambridge: Cambridge University Press.

Mikolov Tomas, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic Regularities in Continuous Space Word Representations. In *Proceedings of NAACL-HLT*, 746–751. http://www.aclweb.org/anthology/N/N13/N13-1090.pdf.

Mikolov, Tomas et al. 2013a. Distributed Representations of Words and Phrases and Their Compositionality. CoRR abs/1310.4546. http://arxiv.org/abs/1310.4546.

Mikolov, Tomas et al. 2013b. Efficient Estimation of Word Representations in Vector Space. CoRR abs/1301.3781. http://arxiv.org/abs/1301.3781.

Muller, Charles. 1964. *Essai de statistique lexicale. L'Illusion Comique de Pierre Corneille*. Paris: Klincksieck.

Muller Charles. 1973. *Initiation aux méthodes de la statistique linguistique*. Paris: Champion.

Muller Charles. 1977. *Principes et méthodes de statistique lexicale*. Paris: Hachette.

Nenadic, Oleg, and Michael Greenacre. 2007. Correspondence Analysis in R, with Two- and Three- Dimensional Graphics: The ca Package. *Journal of Statistical Software* 20 (3): 1–13. http://www.jstatsoft.org.

Oakes, Michael P. 2009. Corpus Linguistics and Language Variation. In *Contemporary Corpus Linguistics*, ed. Paul Baker, 161–185. London: Bloomsbury Continuum.

Paradis, Carita. 1997. *Degree Modifiers of Adjectives in Spoken British English*. Lund: Lund University Press.

Pavlov, Ivan Petrovich. 1927. *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*. London: Oxford University Press/Humphrey Milford.

Pearson, Karl. 1901. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine* 2 (11), 559–572. doi:10.1080/14786440109462720.

Pecina, Pavel. 2010. Lexical Association Measures and Collocation Extraction. *Language Resources and Evaluation* 44 (1): 137–158.

Pedersen, Ted. 1996. Fishing for Exactness. In *Proceedings of the South-Central SAS Users Group Conference*, 188–200. San Antonio, TX: SAS Users Group.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 1532–1543. http://www.aclweb.org/anthology/D14-1162.

Plag, Ingo. 2003. *Word-Formation in English*. Cambridge: Cambridge University Press.

Read, Timothy R.C., and Noel A.C. Cressie. 1988. *Goodness-of-Fit Statistics for Discrete Multivariate Data*. Springer Series in Statistics. New York: Springer-Verlag.

Rescorla, Robert A. 1968. Probability of Shock in the Presence and Absence of CS in Fear Conditioning. *Journal of Comparative and Physiological Psychology* 66: 1–5.

Robinson, David. 2016. *gutenbergr: Download and Process Public Domain Works from Project Gutenberg*. R Package Version 0.1.2. https://CRAN.Rproject.org/package=gutenbergr

Sampson, Geoffrey. 2001. *Empirical Linguistics*. London: Continuum.

Schmid, Hans-Jörg. 2010. Does Frequency in Text Really Instantiate Entrenchment in the Cognitive System? In *Quantitative Methods in Cognitive Semantics: Corpus-Driven Approaches*, ed. Dylan Glynn and Kerstin Fischer, 101–133. Berlin, New York: Mouton de Gruyter.

Schmid, Helmut. 1994. Probabilistic Part-of-Speech Tagging Using Decision Trees. In *International Conference on New Methods in Language Processing*, Manchester, UK, 44–49. http://www.cis.unimuenchen.de/~schmid/tools/TreeTagger/data/treetagger1.pdf.

Sinclair, John. 1966. Beginning the Study of Lexis. In *In Memory of J.R. Firth*, ed. C.E. Bazell, et al., 410–431. London: Longman.

Sinclair, John. 1987. Collocation: A Progress Report. In *Language Topics: Essays in Honour of Michael Halliday*, ed. R. Steele and T. Threadgold, Vol. 2, 19–331. Amsterdam: John Benjamins.

Sinclair, John. 1991. *Corpus, Concordance Collocation*. Oxford: Oxford University Press.

Sinclair, John, and Ronald Carter. 2004. *Trust the Text: Language Corpus and Discourse*. London: Routledge.

Smith, Nicholas. 1997. Improving a Tagger. In *Corpus Annotation: Linguistic Information from Computer Text Corpora*, ed. Roger Garside, Geoffrey N. Leech, and Tony McEnery, 137–150. London: Longman.

Stefanowitsch, Anatol, and Stefan Thomas Gries. 2003. Collostructions: Investigating the Interaction of Words and Constructions. *International Journal of Corpus Linguistics* 8 (2): 209–243.

Stefanowitsch, Anatol, and Stefan Thomas Gries. 2005. Covarying Collexemes. *Corpus Linguistics and Linguistic Theory* 1 (1), 1–46.

Stubbs, Michael. 2001. Words and Phrases: Corpus Studies of Lexical Semantics. Oxford: Wiley–Blackwell.

Tagliamonte, Sali, and Rachel Hudson. 1999. *Be like* et al. Beyond America: The Quotative System in British and Canadian Youth. *Journal of Sociolinguistics* 3 (2): 147–172. doi:10.1111/1467-9481.00070

Tang, Naping, et al. 2009. Coffee Consumption and Risk of Lung Cancer: A Meta-Analysis. *Lung Cancer* 67 (1), 17–22. doi:1.1016/j.lungcan.2009.03.012. http://dx.doi.org/10.1016/j.lungcan.2009.03.012

Taylor, John R. 2012. *The Mental Corpus: How Language is Represented in the Mind*. Oxford: Oxford University Press.

Teubert, Wolfgang. 2005. My Version of Corpus Linguistics. *International Journal of Corpus Linguistics* 10 (1), 1–13. doi:10.1075/ijcl.10.1.01teu.

The Bank of English (1980s–). http://www.titania.bham.ac.uk/. Jointly owned by HarperCollins Publishers and the University of Birmingham.

The British National Corpus. 2007. *BNC XML Edition*. Version 3. http://www.natcorp.ox.ac.uk/. Distributed by Oxford University Computing Services on behalf of the BNC Consortium.

Tognini-Bonelli, Elena. 2001. *Corpus Linguistics at Work*. Amsterdam: John Benjamins.

Toutanova, Kristina, et al. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proceedings of HLTNAACL 2003*, 252–259.

van der Maaten, Laurens, and Geoffrey Hinton. 2008. Visualizing Data Using t-SNE. *Journal of Machine Learning Research* 9: 2579–2605.

Wagner, Allan R., and Robert A. Rescorla. 1972. A Theory of Pavlovian Conditioning: Variations in the Effectiveness of Reinforcement and Nonreinforcement. In *Classical Conditioning II*, ed. Abraham H. Black, and William F. Prokasy, 64–99. New York: Appleton-Century-Crofts.

Ward, Joe H. 1963. Hierarchical Grouping to Optimize an Objective Function. *Journal of the American Statistical Association* 58 (301), 236. ISSN:01621459. http://dx.doi.org/10.2307/2282967.

Wasow, Thomas, and Jennifer Arnold. 2005. Intuitions in Linguistic Argumentation. *Lingua* 115 (11), 1481–1496. http://dx.doi.org/10.1016/j.lingua.2004.07.001.

Wiechmann, Daniel. 2008. On the Computation of Collostruction Strength: Testing Measures of Association as Expressions of Lexical Bias. *Corpus Linguistics and Linguistic Theory* 4 (2): 253.

Yates, Frank. 1984. Tests of Significance for 2×2 Contingency Tables. *Journal of the Royal Statistical Society Series A (General)*, 147 (3): 426–463.

Zeileis, Achim, David Meyer, and Kurt Hornik. 2007. Residual-Based Shadings for Visualizing (Conditional) Independence. *Journal of Computational and Graphical Statistics* 16 (3), 507–525. http://statmathwuacat/~zeileis/papers/Zeileis+Meyer+Hornik-2007.pdf.

Zeldes, Amir. 2012. *Productivity in Argument Selection: From Morphology to Syntax*. Berlin, New York: Mouton de Gruyter.

Zipf, George K. 1949. *Human Behavior and the Principle of Least Effort*. Cambridge: Addison-Wesley.

Solutions

Exercises of Chap. 2

2.1 a.

```
> x <- c(3, 8)
> y <- 2
> z <- c(x,y)
```

`x` is a numeric vector that combines the digits `3` and `8`. `y` is a numeric vector that "contains" `2`. `z` is a numeric vector that combines `3`, `8`, and `2`.

```
> x <- c(3, 7, 9)
> y <- x[3]-x[2]
> z <- y + x[1]
```

Here, `x` is a numeric vector that combines `3`, `7`, and `9`. `y` is a numeric vector that subtracts the second element of `x` (`7`) from the third element of `x` (`9`). If you enter `y`, you should obtain `2` ($9-7$). `z` is a numeric vector that sums `y` (`2`) and the first element of `x` (`3`). The sum of `y` and `z` is therefore `5` ($2+3$).

b. The vector `x` combines three digits that are treated as characters, as evidenced by the presence of quotes. `x[1]` denotes `"3"` and `x[2]` denotes `"7"`. You cannot do maths with characters. The only way to sum 3 and 7 is to declare them as digits in a numeric vector by removing the quotes.

```
> x <- c(3, 7, 9)
> x[1] + x [2]
[1] 10
```

c.

```
> y <- c(1,3,5,5)
> y[c(1,3)]
```

`y` is a numeric vector that combines `1`, `3`, `5`, and `5`. In `y[c(1,3)]`, the square brackets subset the first and third positions from `y`. R should output `1` and `5`.

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8

d.

```
> i <- rep(1,5)
> j <- rep(6,7)
> k <- rep(i, 3)
> g <- c(i,j,k)
```

`i` is a numeric vector that replicates `1` five times. `j` is a numeric vector that replicates `6` seven times. `k` is a numeric vector that replicates `i` (i.e. `1 1 1 1 1`) three times. `g` combines `i` (`1 1 1 1 1`), `j` (`6 6 6 6 6 6 6`), and `k` (`1 1 1 1 1 1 1 1 1 1 1 1 1 1 1`). R should therefore output: `1 1 1 1 1 6 6 6 6 6 6 6 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1`.

e.

```
> ww <- 1+3
```

`ww` is the sum of `1` and `3`. It is a numeric vector of length 1.

```
> xx <- c(ww,4)
```

`xx` combines the vector of length 1 `ww` and `4`. It is a numeric vector of length 2.

```
> yy <- c(xx,8)
```

`yy` combines the vector of length 2 `xx` and `8`. It is a numeric vector of length `3`.

```
> z <- rep(yy,3)
```

`ww` "contains" `4` and `xx` combines `4` and `4`, yielding `4 4`. `yy` combines `4 4` and `8`, yielding `4 8 8`. `z` replicates `yy` three times, yielding `4 8 8 4 8 8 4 8 8`.

f. In general, proceed from the innermost level to the outermost level.

```
> i_1 <- c(2,5,9)
> i_2 <- mean(i_1)
> i_3 <- seq(1,3)
> i_4 <- c(i_3, i_2)
> i <- rep(i_4, 3)
```

g.

```
> integers <- 1:5
> five_letters <- c("a","b","c","d","e")
> names(integers) <- five_letters
> integers
```

or, shorter:

```
> integers <- 1:5
> names(integers) <- letters[1:5]
```

h.

```
> rep(c("a", "b", "c"), 2)
[1] "a" "b" "c" "a" "b" "c"
```

```
> c(rep(c(T, F, T), 2), rep(T, 5))
 [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> # or
> c(T,F,T,T,F,rep(T,6))
 [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

i. In R, characters are not just letters. They include spaces and punctuation. In `R is great!`, there are eight letters, two spaces, and one exclamation mark. There are therefore 11 characters.

2.2 a.

```
> values <- c(33, 65, 17, 3638)
> row_names <- c("sure", "other adjectives")
> col_names <- c("hell", "other NPs")
> mat <- matrix(values, 2, 2, dimnames=list(row_names, col_names)) ; mat
                 hell other NPs
sure               33        17
other adjectives   65      3638
```

b.

```
> rowSums(mat)
            sure other adjectives
              50             3703
> colSums(mat)
     hell other NPs
       98      3655
```

c.

```
> source("/CLSR/mypercentage.r") # load the mypercentage() function
> percentage (mat[1,1], sum(mat))
33 is 0.88 percent of 3753
```

2.3 a.

```
> rank <- 1:5
> A <- c("good", "quick", "right", "large", "safe")
> NP <- c("gold", "flash", "rain", "life", "houses")
> G2 <- c(288.82, 189.29, 175.98, 164.55, 148.32)
> df <- data.frame(rank, A, NP, G2)
> df
  rank     A     NP     G2
1    1  good   gold 288.82
2    2 quick  flash 189.29
3    3 right   rain 175.98
4    4 large   life 164.55
5    5  safe houses 148.32
```

b.

```
> mean(df$G2)
[1] 193.392
```

c.

```
> mean(df$G2[c(1:3)])
[1] 218.03
> # or
> mean(df$G2[1:3])
[1] 218.03
```

d.

```
> sort(df$A)
[1] good  large quick right safe
Levels: good large quick right safe
```

e.

```
> sort(df$NP, decreasing=TRUE)
[1] rain   life   houses gold   flash
Levels: flash gold houses life rain
```

Exercise of Chap. 3

Visit http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/. Download and install the German parameter files and German chunker parameter files. Open a terminal window and type `sh install-tagger.sh` to run the installation script in the directory where you have downloaded the files (on my computer: `/TreeTagger`, which I access by entering `cd /TreeTagger` in the terminal.)

Next, download the German model from http://datacube.wu.ac.at.

```
> install.packages("openNLPmodels.de", repos = "http://datacube.wu.ac.at/", type = "source")
```

Load `NLP`, `openNLP`, and `openNLP.de`.

```
> library(NLP)
> library(openNLP)
> library(openNLPmodels.de)
```

For the function to run properly, you need to set the `language` argument of both

`Maxent_Word_Token_Annotator()` and
`Maxent_POS_Tag_Annotator()`

to `de` (the ISO code for German).

```
> tagPOS <-  function(x, ...) {
+   s <- as.String(x)
+   word_token_annotator <- Maxent_Word_Token_Annotator(language="de")
+   a2 <- Annotation(1L, "sentence", 1L, nchar(s))
+   a2 <- annotate(s, word_token_annotator, a2)
+   a3 <- annotate(s, Maxent_POS_Tag_Annotator(language="de"), a2)
+   a3w <- a3[a3$type == "word"]
+   POStags <- unlist(lapply(a3w$features, `[[`, "POS"))
+   POStagged <- paste(sprintf("%s_%s", s[a3w], POStags), collapse = " ")
+   list(POStagged = POStagged, POStags = POStags)
+ }
```

The unique argument of `TagPOS()` is the text to be tagged.

```
> text <- scan("/CLSR/chap3/exercise/Brot.txt", what="char", sep="\n")
> tagged_txt <-  tagPOS(text)
```

To print the tagged text only, access the `POStagged` element of `tagged_txt`.

```
> tagged_txt$POStagged
```

This is what you should obtain:

```
Brot_NN ist_VAFIN ein_ART traditionelles_ADJA
Nahrungsmittel_NN ,_$, das_PRELS aus_APPR einem_ART
Teig_NN aus_APPR gemahlenem_ADJA Getreide_NN (_$(
Mehl_NE )_$( ,_$, Wasser_NN ,_$, einem_ART Triebmittel_NN
und_KON meist_ADV weiteren_ADJA Zutaten_NN gebacken_VVINF
wird_VAFIN ._$.  Es_PPER zählt_VVFIN zu_APPR den_ART
Grundnahrungsmitteln_NN ._$.  Das_ART feste_ADJA ,_$,
dunkle_ADJA Äußere_NN des_ART Brotes_NN heißt_VVFIN
Kruste_NN oder_KON Rinde_NN ._$.  Das_ART Innere_NN
ist_VAFIN die_ART Krume_NN ._$.  Brotkrümel_NN
heißen_VVFIN auch_ADV Brosamen_NN (_$( aus_APPR dem_ART
Mittelhochdeutschen_NN )_$( oder_KON Brösel_NE .
_$.  Die_ART meisten_PIAT Brotteige_NN können_VVFIN
auch_ADV in_APPR Form_NN kleinerer_ADJA ,_$, etwa_ADV
handtellergroßer_ADJA Portionen_NN als_APPR Brötchen_NN
gebacken_VVPP werden_VAINF ._$.
```

Exercises of Chap. 4

4.1 The first solution consists in looking for a character string that contains `"c"` or `"m"` followed by `"at"`.

```
> catmat <- c("the", "cat", "sat", "on", "the", "mat")
> grep("[cm]at", catmat, value=TRUE) # solution 1
[1] "cat" "mat"
```

The second solution consists in looking for a character string that does not contain `"s"` followed by `"at"`.

```
> grep("[^s]at", catmat, value=TRUE) # solution 2
[1] "cat" "mat"
```

4.2 Clear R's memory and load the text.

```
> rm(list=ls(all=TRUE))
> text <- scan("C:/CLSR/chap4/gnu.txt", what="char", sep="\n") # Windows
> text <- scan("/CLSR/chap4/gnu.txt", what="char", sep="\n") # Mac
```

Next, split the character vector `text` at non-word characters with `strsplit()`. Do not forget to unlist the output of `strsplit()` with `unlist()`.

```
> text.split <- unlist(strsplit(text, "\\W+"))
```

Look for the occurrences of *the* with `grep()`. Make sure that you enclose your match within `^` and `$` so that only *the* as a stand-alone word is retrieved. See what happens when you do not

```
> grep("the", text.split, value=TRUE)
 [1] "other" "other" "the"   "the"   "the"   "the"   "other" "the"   "them"  "the"   "these"
```

R retrieves all occurrences of *the*, including when it is part of a larger word. Another issue that you have to deal with is case sensitivity. In your text vector, the definite article comes in two forms: *The* and *the*. For R to recognize the former, set the `ignore.case` argument to `TRUE`.

```
> grep("^the$", text.split, value=TRUE) # case sensitive
[1] "the" "the" "the" "the" "the" "the"
> grep("^the$", text.split, ignore.case=TRUE, value=TRUE) # case insensitive
[1] "The" "The" "the" "the" "the" "the" "the" "the"
```

All that is left to do is count the number of matches with `length()`.

```
> the.in.text.split <- grep("^the$", text.split, ignore.case=TRUE, value=TRUE)
> length(the.in.text.split)
[1] 8
```

There are eight occurrences of *the* in the text.

4.3 First, load the corpus file.

```
> rm(list=ls(all=TRUE))
> corp.file <- scan("/CLSR/chap4/102CTL002_annotated.txt", what="char", sep="\n")
```

a. Because this is an exact matching task, you will need `strapply()` from the `gsubfn` package. According to the CLAWS7 tagset, the past participle form of a lexical verb is coded `VVN`. Therefore, we look for one or more characters followed by an underscore and the desired tag: `\\w+_VVN`. By placing the character string between brackets, we make sure that the tag is not retrieved because `strapply()` applies backreferencing by default (as if we had `backref=-1`). If we did want the tag, we would set `backref` to 0.

```
> library(gsubfn)

Loading required package:  proto

> unlist(strapply(corp.file, "(\\w+)_VVN")) # without the tag
[1] "doubled"      "caused"       "provided"     "acknowledged"
> unlist(strapply(corp.file, "(\\w+)_VVN", backref=-1)) # without the tag
[1] "doubled"      "caused"       "provided"     "acknowledged"
> unlist(strapply(corp.file, "(\\w+)_VVN", backref=0)) # with the tag
[1] "doubled_VVN"      "caused_VVN"       "provided_VVN"     "acknowledged_VVN"
```

b. To retrieve all the verb forms of *be*, we need to write a regular expression that encompasses the following tags:

- `VBDR`: *were*;
- `VBDZ`: *was*;
- `VBG`: *being*;
- `VBI`: *be*;
- `VBM`: *am*;
- `VBN`: *been*;

- VBR: *are*;
- VBZ: *is*.

All the tags have VB in common. One safe but clumsy way of retrieving the desired verbs forms is to look for VB followed by D, or G, or I, or M, or N, or R, or Z, and optionally by R or Z (to match VBDR and VBDZ).

```
> unlist(strapply(corp.file, "(\\w+)_VB[DGIMNRZ][RZ]*", backref=-1)) # without the tag
[1] "be"  "is"  "is"  "are" "is"  "be"
```

A far more elegant query involves VB followed by one or two word characters.

```
> unlist(strapply(corp.file, "(\\w+)_VB\\w{1,2}", backref=-1)) # without the tag
[1] "be"  "is"  "is"  "are" "is"  "be"
```

c. To retrieve all the 3SG forms of *be*, *do*, and *have* in the simple present, we need to write a regular expression that encompasses the following tags:

- VBZ: *is*;
- VDZ: *does*;
- VHZ: *has*.

All the tags have V.Z in common. They differ with respect to the middle letter: B, D, or H. This is how we translate the query using a regular expression:

```
> unlist(strapply(corp.file, "(\\w+)_V[BDH]Z", backref=-1)) # without the tag
[1] "is"  "is"  "has" "is"
```

There is no 3SG form of *do* in the corpus file.

Exercises of Chap. 5

5.1 Clear R's memory, and install and load the gutenbergr package.

```
> rm(list=ls(all=TRUE))
> install.packages("gutenbergr")
> library(gutenbergr)
```

Download the three novels and inspect.

```
> dickens <- gutenberg_download(c(1400, 1023, 766))

Determining mirror for Project Gutenberg from http://www.gutenberg.org/robot/harvest
Using mirror http://aleph.gutenberg.org
```

```
> str(dickens)
Classes 'tbl_df', 'tbl' and 'data.frame': 98053 obs. of  2 variables:
 $ gutenberg_id: int  766 766 766 766 766 766 766 766 766 766 ...
 $ text        : chr  "DAVID COPPERFIELD" "" "" "By Charles Dickens" ...
```

dickens is a two-column data frame with one row for each line of the texts.[1] The two columns are:

- gutenberg_id: a column of integers with the Project-Gutenberg ID of each text;
- text: a character vector containing the lines of the text(s).

[1] More specifically, according to the documentation, it is a two-column "tbl_df ", as used by the tibble and dplyr packages.

Create an empty text file to store the concordance and add the column headers manually.

```
> output.conc <- "C:/CLSR/chap5/conc.exercise.txt" # Windows
> output.conc <- "/CLSR/chap5/conc.exercise.txt" # Mac
> cat("PROJECT GUTENBERG ID\tLEFT CONTEXT\tNODE\tRIGHT CONTEXT\n", file=output.conc)
```

Vectorize the match and select the contextual window.

```
> match <- "London"
> context <- 10
```

In Sect. 5.2.1.4, we used one loop to iterate over the matches. In this exercise, you need to embed that loop into another loop, which iterates over the three novels. In this embedding loop, the variable `i` stands for the ID of each novel successively.

```
> # enter the first loop
> for (i in unique(dickens$gutenberg_id)){ # access each novel by ID
+   # split at non-word characters
+   split <- unlist(strsplit(dickens$text[which(dickens$gutenberg_id==i)], "\\W+"))
+   # find the positions of the match/node
+   node.positions <- grep(match, split)
+
+     # enter the second loop
+     for (j in 1:length(node.positions)){ # access each match...
+         # access the current match
+       node <- split[node.positions[j]]
+         # access the left context of the current match
+         if (node.positions[j] < context)
+         if (node.positions[j]==1) left.context <- NULL
+         else left.context <- split[1:(node.positions[j]-1)]
+         else left.context <- split[(node.positions[j]-context):(node.positions[j]-1)]
+
+         # access the right context of the current match
+         if ((node.positions[j] + context) >= length(split))
+         if (node.positions[j]==length(split)) right.context <- NULL
+         else right.context <- split[(node.positions[j]+1):length(split)]
+         else right.context <- split[(node.positions[j]+1):(node.positions[j]+context)]
+
+         # print the results
+         cat(i,"\t", left.context,"\t", node, "\t", right.context, "\n", file=output.conc, append=TRUE)
+     } # exit the second loop
+ } # exit the first loop
```

Tab. B.1 displays a sample of the output.

5.2 The first part of the script from Sect. 5.3.3 does not change.

```
> # Clear R's memory
> rm(list=ls(all=TRUE))
>
> # load the gsubfn package
> library(gsubfn)
>
> # load the paths to all the corpus files
> corpus.files <- list.files(path="/CLSR/BNC_baby", pattern="\\.xml$", full.names=TRUE)
```

The expression for the match needs to be formulated so as to retrieve all instances of the *it BE ADJ to V that* construction. It contains a sequence of the following elements:

- the pronoun *it*:
  `<w c5=\"PNP\" hw=\"it\" pos=\"PRON\">it </w>`

Table B.1: A concordance of *London* in three novels by C. Dickens (excerpt)

PG ID	LEFT CONTEXT	NODE	RIGHT CONTEXT
1023	Bleak House true There is in that city of	London	there some property of ours which is much at this
766	Martha wants she said to Ham to go to	London	Why to London returned Ham He stood between them
1023	Oh dear no miss he said This is a	London	particular I had never heard of such a thing
1400	uns if you please good Lord and not my	London	gentleman No no We ll show em another pair of
766	a month and his sister and my aunt came to	London	to see him Agnes and I parted from him aboard
1023	a more convenient place for all of us So to	London	we will go That being settled there is another thing
1400	a walk and that I would go on along the	London	road while Mr Jaggers was occupied if he would let
766	affect them who were so innocent of London life and	London	streets to discover how knowing I was and was ashamed
1400	after day a vast heavy veil had been driving over	London	from the East and it drove still as if in
766	Agnes Well I returned See here You come to	London	I rely on you and I have an object and
…	…	…	…

- all the forms of the copula *be*:
  `<w c5=\"VB[BDGINZ]\" hw=\"be\" pos=\"VERB\">\\w+ </w>`
- any adjective:
  `<w c5=\"AJ0\" hw=\"\\w+\" pos=\"ADJ\">\\w+ </w>`
- the infinitive marker *to*:
  `<w c5=\"TO0\" hw=\"to\" pos=\"PREP\">to </w>`
- any verb in the infinitive:
  `<w c5=\"VVI\" hw=\"\\w+\" pos=\"VERB\">\\w+ </w>`
- the conjunction *that*:
  `<w c5=\"CJT\" hw=\"that\" pos=\"CONJ\">that </w>`

Note that all quotes must escaped so as not to be mixed up with R's character-string delimiter.

```
> # vectorize the search expression
> expr.match <- "<w c5=\"PNP\" hw=\"it\" pos=\"PRON\">it </w>
+ <w c5=\"VB[BDGINZ]\" hw=\"be\" pos=\"VERB\">\\w+ </w>
+ <w c5=\"AJ0\" hw=\"\\w+\" pos=\"ADJ\">\\w+ </w>
+ <w c5=\"TO0\" hw=\"to\" pos=\"PREP\">to </w>
+ <w c5=\"VVI\" hw=\"\\w+\" pos=\"VERB\">\\w+ </w>
+ <w c5=\"CJT\" hw=\"that\" pos=\"CONJ\">that </w>"
```

Prepare an empty vector to collect all matches at the end of the first for loop

```
> all.matches <- character()
```

Before entering the loop, you need to change the relevant elements to be collected, namely `ADJ` and `V_inf`.

```
>   ADJ <- unlist(strapplyc(matches, "pos=\"ADJ\">(\\w+) </w>"))
>   V_inf <- unlist(strapplyc(matches, "<w c5=\"VVI\" hw=\"(\\w+)\"", backref=1))
```

Here is what the full loop looks like.

```
> # enter the loop
> for (i in 1:length(corpus.files)) {
+   # load current corpus file
+   corpus.file <- scan(corpus.files[i], what="char", sep="\n")
+   classcode <- unlist(strapplyc(corpus.file, "<classCode scheme=\"DLEE\">(.*?)</classCode>"))
```

```
+   # retrieve corpus file info
+   info <- unlist(strapplyc(corpus.file, "<[ws]text type=\"\\w+\">", backref=1))
+   # look for mode
+   mode <- unlist(strapplyc(info, "[ws]text"))
+   # look for type
+   type <- unlist(strapplyc(info, "[ws]text type=\"(\\w+)\""))
+
+   # isolate corp sentences and discard header
+   sentences <- grep("<s n=", corpus.file, perl=TRUE, value=TRUE)
+   # get matches (full pattern)
+   matches <- unlist(strapplyc(sentences, expr.match, ignore.case=TRUE, backref=0))
+   # if there are no matches, go to next corpus path ...
+   if (length(matches)==0) { next }
+
+   # collect the relevant elements
+   ADJ <- unlist(strapplyc(matches, "pos=\"ADJ\">(\\w+) </w>"))
+   V_inf <- unlist(strapplyc(matches, "<w c5=\"VVI\" hw=\"(\\w+)\"", backref=1))
+
+   # clean up
+   matches.clean <- gsub("<.*?>", "", matches, perl=T) # remove xml tags
+   matches.clean <- gsub("< +$>", "", matches.clean, perl=T) # remove trailing spaces
+
+   # paste the collected elements
+   matches.row <- paste(basename(corpus.files[i]), classcode, mode, type, matches.clean,
+                        ADJ, V_inf, sep="\t")
+
+   # collects all results
+   all.matches <- c(all.matches, matches.row)
+ } # exit the loop
```

Finally, save the results into a text file. Do not forget to add tab-separated column headers.

```
> cat("corpus file\tinfo\tmode\ttype\texact match\tADJ\tV_inf", all.matches, sep="\n",
+     file="/CLSR/chap5/df_exercise.txt")
```

Tab. B.2 displays a sample of the output.

5.3 Clear R's memory, load `gutenbergr`, download the novel (whose ID is 2701), and inspect `moby`.

```
> rm(list=ls(all=TRUE))
> library(gutenbergr)
> moby <- gutenberg_download(2701) # load Moby Dick from Project Gutenberg
> head(moby)
# A tibble: 6 x 2
  gutenberg_id           text
         <int>          <chr>
1         2701     MOBY DICK;
2         2701
3         2701
4         2701 or, THE WHALE.
5         2701
6         2701
> tail(moby)
# A tibble: 6 x 2
  gutenberg_id                                                                text
         <int>                                                               <chr>
1         2701 and night, I floated on a soft and dirgelike main. The unharming sharks,
2         2701 they glided by as if with padlocks on their mouths; the savage sea-hawks
3         2701 sailed with sheathed beaks. On the second day, a sail drew near, nearer,
4         2701    and picked me up at last. It was the devious-cruising Rachel, that in
5         2701      her retracing search after her missing children, only found another
6         2701                                                              orphan.
> str(moby)
Classes 'tbl_df', 'tbl' and 'data.frame': 22258 obs. of  2 variables:
 $ gutenberg_id: int  2701 2701 2701 2701 2701 2701 2701 2701 2701 2701 ...
 $ text        : chr  "MOBY DICK;" "" "" "or, THE WHALE." ...
```

Table B.2: A data frame of the *it BE ADJ to V that* construction in the BNC baby (sample)

corpus file	info	mode	type	exact match	ADJ	V_inf
A1J.xml	W newsp brdsht nat: report	wtext	NEWS	it is logical to assume that	logical	assume
A1N.xml	W newsp brdsht nat: sports	wtext	NEWS	It is fair to say that	fair	say
A6U.xml	W ac:humanities arts	wtext	ACPROSE	It is easy to say that	easy	say
A98.xml	W newsp brdsht nat: social	wtext	NEWS	It is interesting to note that	interesting	note
ACJ.xml	W ac:polit law edu	wtext	ACPROSE	it is fair to say that	fair	say
ACJ.xml	W ac:polit law edu	wtext	ACPROSE	it is unacceptable to claim that	unacceptable	claim
ACJ.xml	W ac:polit law edu	wtext	ACPROSE	it is possible to concede that	possible	concede
AHC.xml	W newsp brdsht nat: misc	wtext	NEWS	it is hard to believe that	hard	believe
AJF.xml	W newsp brdsht nat: arts	wtext	NEWS	it is good to report that	good	report
AL2.xml	W newsp brdsht nat: commerce	wtext	NEWS	it is possible to argue that	possible	argue
...	...	...	...	...	...	...

We are only interested in the `text` component of `moby`. We isolate this component.

```
> moby <- moby$text
```

Time for some cleanup. First, switch the whole novel to lowercase, so as not to alter the count later on.

```
> moby <- tolower(moby)
```

Next, remove everything that is not a letter or a space. You do it by substituting whatever is not a letter or a space by a space. To understand the code below, remember that `[^xyz]` means "not `x`, or `y`, or `z`".

```
> moby <- gsub("[^a-zA-Z\\s]", " ", moby); head(moby)
[1] "moby dick "     ""               ""               "or  the whale " ""
[6] ""
```

The above line of code is very likely to generate multiple spaces. You remove them after getting rid of leading and trailing spaces.

```
> moby <- gsub("(^ | +$)", "", moby) # remove leading/trailing spaces
> moby <- gsub(" +", " ", moby) # remove multiple spaces
```

At this stage, your `moby` vector is full of empty strings. Get rid of them via subsetting.

```
> moby <- moby[which(moby!="")]
> head(moby)
[1] "moby dick"                                  "or the whale"
[3] "by herman melville"                         " contents"
[5] " etymology"                                 " extracts supplied by a sub sub librarian"
```

Your character vector consists of as many elements as the novel has lines (enter `length(moby)` to obtain the number of lines). To proceed, `moby` must be of length 1. Gather all the elements of `moby` into one element with `paste()`. The value of the `collapse` argument is a space, to delimit the conflated elements.

```
> moby.one <- paste(moby, collapse=" ")
```

It is now time to generate bigrams. This is easy. Load the `ngram` package and use the dedicated function: `ngram()`. The main argument is the character string that serves as input (in your case `moby.one`). The second argument is the value of *n* (2 for bigrams, 3 for trigrams, etc.).

```
> library(ngram)
> text.bigrams <- ngram(moby.one, n=2)
> text.bigrams
An ngram object with 116430 2-grams
```

According to the package documentation, you access the list thanks to the `get.phrasetable()` function.

```
> bigrams <- get.phrasetable(text.bigrams)
> head(bigrams)
      ngrams freq        prop
1    of the  1886 0.008610142
2    in the  1184 0.005405307
3    to the   732 0.003341794
4  from the   441 0.002013294
5 the whale   414 0.001890031
6    of his   373 0.001702854
```

The output is a data frame with three columns: `ngram`, `freq`, and `prop`. Get rid of the third column.

```
> bigrams$prop <- NULL
```

The good news is that the frequency list is almost done, thanks to `get.phrasetable()`. The bad news is that you need to split the `ngram` column in two to apply the stoplist. Fortunately, there are many easy ways of doing it. The solution below hinges on `do.call()`. The function executes `rbind()` to make a new data frame where each line of `bigrams$ngram` is split in two by means of `strsplit()`. A third column, `bigrams$freq` is appended to this data frame.

```
> bigrams <- data.frame(do.call('rbind', strsplit(as.character(bigrams$ngram), " ")), bigrams$freq)
> colnames(bigrams)[1] <- "word.1" # change name of 1st column
> colnames(bigrams)[2] <- "word.2" # change name of 2nd column
> str(bigrams)
'data.frame': 116430 obs. of  3 variables:
 $ word.1      : Factor w/ 16956 levels "a","aback","abaft",..: 10047 7448 15089 5986 14859 10047
    489 10102 10047 848 ...
 $ word.2      : Factor w/ 16957 levels "a","aback","abaft",..: 14860 14860 14860 14860 16535 6982
    14860 14860 1 14860 ...
 $ bigrams.freq: int  1886 1184 732 441 414 373 372 359 334 330 ...
```

You are almost there. Load the stoplist used in the Chap. 5.

```
> stoplist <- scan("/CLSR/chap5/english_stoplist.txt", what="character", sep="\n")
```

You want the stoplist to apply to the words found in each column. You subset the lines in the `bigrams` data frame whose words are not those found in the stoplist.

```
> bigrams.filtered <- bigrams[which(!bigrams$word.1 %in% stoplist & !bigrams$word.2 %in% stoplist),]
> head(bigrams.filtered, 10)
     word.1 word.2 bigrams.freq
27    sperm  whale          183
72    white  whale          106
99     moby   dick           85
152 captain   ahab           62
216    mast   head           50
337    mast  heads           37
377   whale   ship           34
406 captain  peleg           32
```

```
415 quarter   deck          32
420     aye    aye          32
```

In case you still do not know what the novel is about, the most frequent bigrams are *sperm whale*, *white whale*, and *Moby Dick*.

Exercises of Chap. 6

6.1 Clear R's memory and load the data frame.

```
> rm(list=ls(all=TRUE))
> df <- readRDS("/CLSR/chap6/modals.by.genre.BNC.rds")
> str(df)
'data.frame': 10 obs. of  8 variables:
 $ ACPROSE : int  44793 17379 35224 11293 13511 976 4097 19420 32805 29903
 $ CONVRSN : int  23161 7955 628 3524 2989 451 1639 4344 9032 9895
 $ FICTION : int  32293 49826 5302 13917 15043 1649 4855 13791 24285 56934
 $ NEWS    : int  16269 14045 6134 3634 4306 221 408 8900 37476 23375
 $ NONAC   : int  53297 32923 32934 13110 15522 1115 3746 25622 53246 59211
 $ OTHERPUB: int  53392 19684 20923 7215 11176 477 2306 22014 66980 33132
 $ OTHERSP : int  26262 11976 4267 4710 3045 820 1233 7647 15934 23778
 $ UNPUB   : int  11816 5733 6853 1465 4064 110 1701 7104 19258 8741
```

Because you are making barplots, you need to convert the data frame into a matrix.

```
> mat <- as.matrix(df)
```

Before entering the loop, you need to set up the plotting window so that it can hold ten barplots. Here, we tell R to arrange the ten barplots in two rows and five columns.

```
> par(mfrow=c(2,5))
```

The loop tells R to access each row of the matrix (`i` takes the value of each row). For each row, R makes a barplot of the values that it finds there. Thanks to the argument `las=2`, the labels are perpendicular to *x* axis (by default, they are parallel, because `las=0`). The `main` argument selects an overall title for each plot. This title is the name of each row, i.e. the corresponding English modal. For a change, the columns will be in Dodger blue.

```
> for (i in 1:nrow(mat)){ # for each row in the matrix
+   barplot(mat[i,], las=2, main=rownames(mat)[i], col="dodgerblue") # make a barplot
+ }
```

Fig. B.1 shows what the loop outputs. Be careful when you interpret a barplot. The height of each bar should be interpreted in the light of the size of each text genre. Because the size of CONVRSN is smaller than the size of most of the other genres, you cannot conclude that English modals are used less in conversation.

6.2 Clear R's memory, load the novel using the `gutenbergr` package (see Sol. 5.1 above), and isolate the text.

```
> rm(list=ls(all=TRUE))
> library(gutenbergr)
> ot <- gutenberg_download(730)
> ot <- ot$text
```

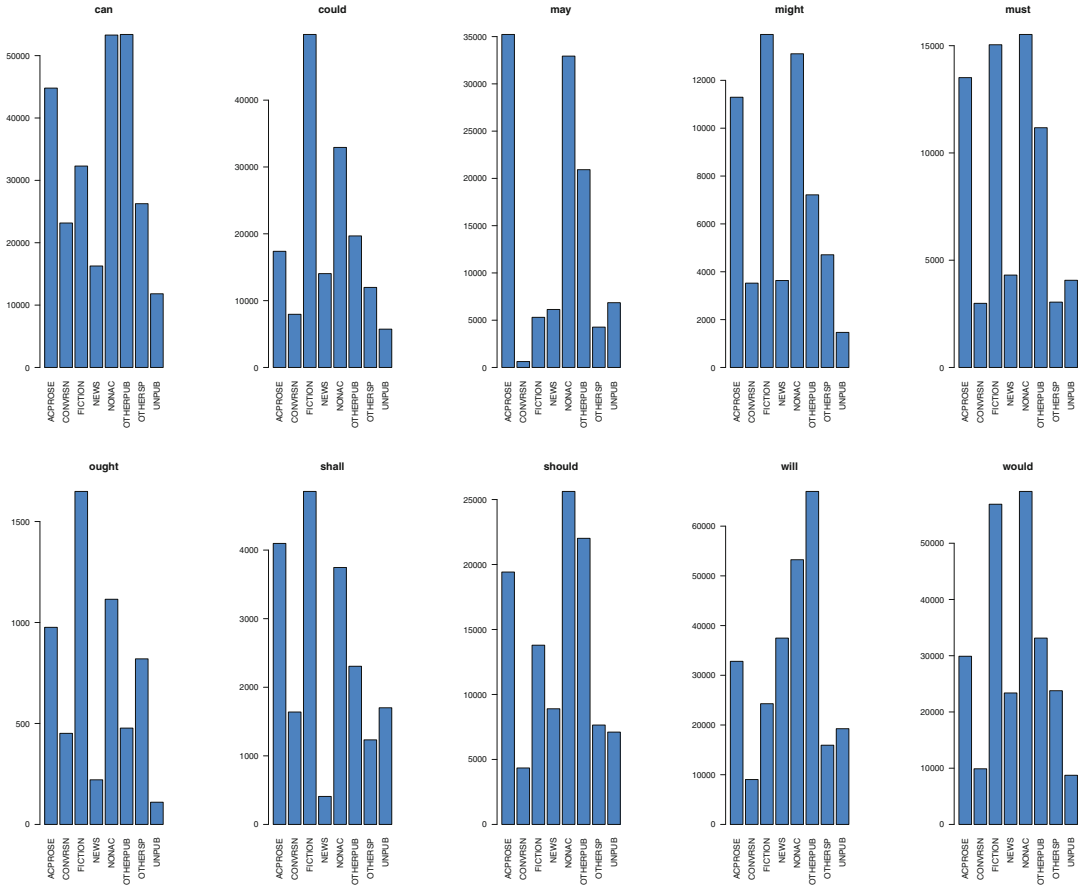


Fig. B.1: A barplot of the distribution of ten English modals across text genres in the BNC

Because we are going to split the text at non-word characters, we replace the spaces with an underscore in the place names that consist of multiple words: *Snow Hill*, *Jacob's Island*, and *London Bridge*. By doing so, we make sure that multiple-word place names are not affected by the split.

```
> ot <- gsub("Snow Hill", "Snow_Hill", ot)
> ot <- gsub("Jacob's Island", "Jacob_s_Island", ot)
> ot <- gsub("London Bridge", "London_Bridge", ot)
```

Next, we split the text and remove the empty character strings.

```
> ot.split <- unlist(strsplit(ot, "\\W+"))
> ot.split <- ot.split[nchar(ot.split)>0]
```

In dispersion plots, the *x*-axis denotes the sequence of words in the novel. We create these values with `seq()` and store them in `seq.corpus`

```
> seq.corpus <- seq(1:length(ot.split))
```

and we vectorize the matches.

```
> matches <- c("Clerkenwell", "Snow_Hill", "Holborn", "Jacob_s_Island",
+              "Southwark", "London_Bridge", "Newgate", "Pentonville", "Smithfield")
```

We need to loop over the matches to generate a dispersion plot for each of them. The loop does the following:

- get the position of the current match in the text;
- replace each position in the text with `NA`;
- replace each `NA` with `1`;
- generate the plot.

Before running the loop, specify the desire plot layout. You obtain Fig. B.2.

```
> par(mfrow=c(5,2))
> for (i in 1:length(matches)) {
+   match.positions <- which(ot.split==matches[i])
+   match.count <- rep(NA, length(seq.corpus))
+   match.count[match.positions] <- 1
+   plot(match.count, xlab="novel", ylab=matches[i], type="h", ylim=c(0,1), yaxt="n")
+ }
```

6.3 Clear R's memory and load the data frame.

```
> rm(list=ls(all=TRUE))
> df <- readRDS("/CLSR/chap6/modals.by.genre.BNC.rds")
> str(df)
'data.frame': 10 obs. of  8 variables:
 $ ACPROSE : int  44793 17379 35224 11293 13511 976 4097 19420 32805 29903
 $ CONVRSN : int  23161 7955 628 3524 2989 451 1639 4344 9032 9895
 $ FICTION : int  32293 49826 5302 13917 15043 1649 4855 13791 24285 56934
 $ NEWS    : int  16269 14045 6134 3634 4306 221 408 8900 37476 23375
 $ NONAC   : int  53297 32923 32934 13110 15522 1115 3746 25622 53246 59211
 $ OTHERPUB: int  53392 19684 20923 7215 11176 477 2306 22014 66980 33132
 $ OTHERSP : int  26262 11976 4267 4710 3045 820 1233 7647 15934 23778
 $ UNPUB   : int  11816 5733 6853 1465 4064 110 1701 7104 19258 8741
```

Because the modals are the row variables and the text genres column variables, you need to transpose the data frame with `t()`. Because the transposition outputs a matrix, convert it back into a data frame.

```
> df <- t(df)
> class(df)
[1] "matrix"
> df <- as.data.frame(df)
```

It is now time to run `stripchart()`. The chart consists of ten strips (one for each modal). For ease of reading, we assign one color per modal with `rainbow()`. The numeric argument of this function specifies how many distinct colors we want. The `pch` argument determines how the points are drawn in the plot (`pch=16` selects a filled circle).[2] Thanks to the `las` argument (`las=1`), the labels are perpendicular to the *y* axis. You should obtain Fig. B.3.

```
> stripchart(df, xlab="frequencies", col = rainbow(8), pch=16, las=1)
```

[2] Enter `?pch` for more details.

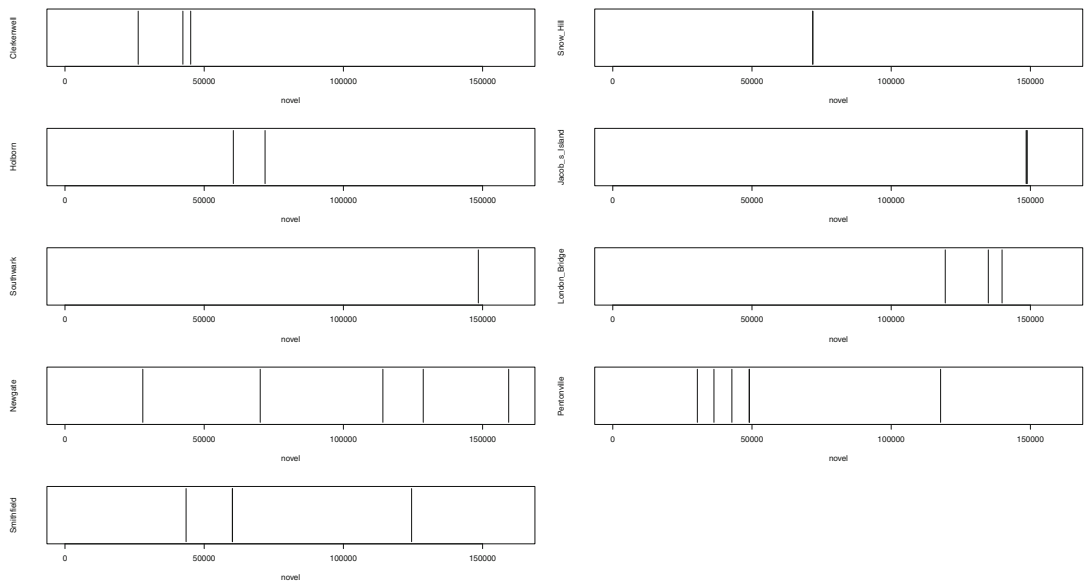


Fig. B.2: Dispersion plots of nine place names in *Oliver Twist*

Exercises of Chap. 7

7.1 Clear R's memory and load the data file.

```
> rm(list=ls(all=TRUE))
> df <- readRDS("/CLSR/chap6/modals.by.genre.BNC.rds")
> str(df)
'data.frame': 10 obs. of  8 variables:
 $ ACPROSE : int  44793 17379 35224 11293 13511 976 4097 19420 32805 29903
 $ CONVRSN : int  23161 7955 628 3524 2989 451 1639 4344 9032 9895
 $ FICTION : int  32293 49826 5302 13917 15043 1649 4855 13791 24285 56934
 $ NEWS    : int  16269 14045 6134 3634 4306 221 408 8900 37476 23375
 $ NONAC   : int  53297 32923 32934 13110 15522 1115 3746 25622 53246 59211
 $ OTHERPUB: int  53392 19684 20923 7215 11176 477 2306 22014 66980 33132
 $ OTHERSP : int  26262 11976 4267 4710 3045 820 1233 7647 15934 23778
 $ UNPUB   : int  11816 5733 6853 1465 4064 110 1701 7104 19258 8741
```

You need to transpose the data frame, otherwise you will calculate the measures of central tendency of text genres, not modals. The transposition process converts the data frame into a matrix, which is fine because we are doing arithmetics.

```
> mat <- t(df)
> str(mat)
 int [1:8, 1:10] 44793 23161 32293 16269 53297 53392 26262 11816 17379 7955 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:8] "ACPROSE" "CONVRSN" "FICTION" "NEWS" ...
  ..$ : chr [1:10] "can" "could" "may" "might" ...
```

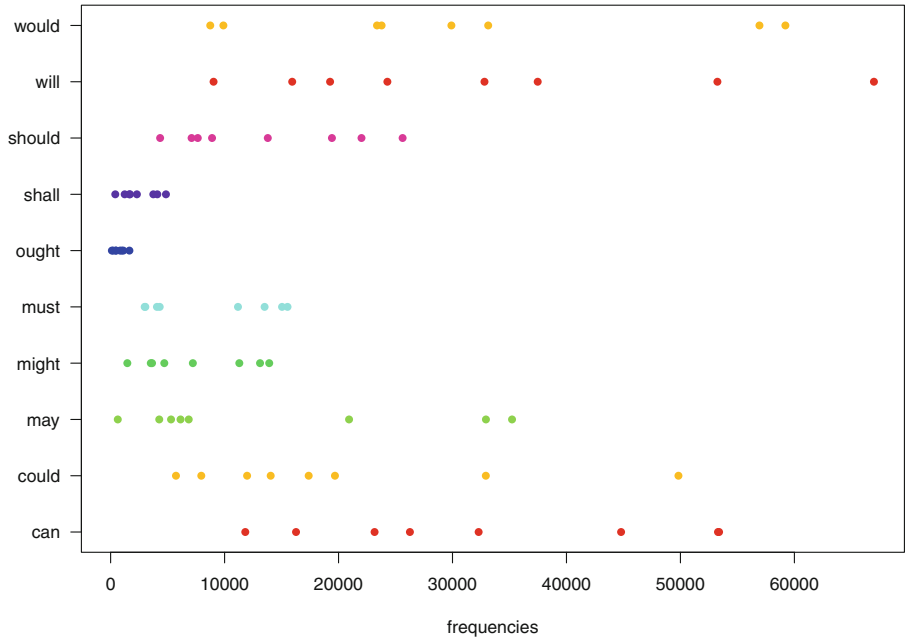


Fig. B.3: A strip chart of the distribution of ten English modals across text genres in the BNC

With a `for` loop, tell R to go over each row and calculate the mean, the median, and find the frequency of the mode. With `cat()`, prefix the row name and append the results, separated by a space. Each line of the output ends with a newline (`"\n"`).

```
> for (i in 1:nrow(mat)){ # for each row in the matrix
+   mean <- mean(mat[i,]) # calculate the mean
+   median <- median(mat[i,]) # calculate the median
+   mode <- max(mat[i,]) # find the frequency of the mode
+   cat(rownames(mat)[i], mean, median, mode, "\n", sep=" ") # print the results
+ } # exit the loop
ACPROSE 20940.1 18399.5 44793
CONVRSN 6361.8 3934 23161
FICTION 21789.5 14480 56934
NEWS 11476.8 7517 37476
NONAC 29072.6 29272.5 59211
OTHERPUB 23729.9 20303.5 66980
OTHERSP 9967.2 6178.5 26262
UNPUB 6684.5 6293 19258
```

7.2 Clear R's memory and create two numeric vectors with the values contained in Tab. 7.1.

```
> rm(list=ls(all=TRUE))
> vg1 <- c(314, 299, 401, 375, 510, 660, 202)
> vg2 <- c(304, 359, 357, 342, 320, 402, 285)
```

Summarizing the data in a plot such as Fig. 7.6 requires several steps. First, create a vector with the days of the week. Assign the names to each element of `vg1` and `vg2`.

```
> days <- c("Mon","Tue","Wed","Thur","Fri","Sat","Sun")
> names(vg1) <- days
> names(vg2) <- days
> vg1
 Mon  Tue  Wed Thur  Fri  Sat  Sun
 314  299  401  375  510  660  202
> vg2
 Mon  Tue  Wed Thur  Fri  Sat  Sun
 304  359  357  342  320  402  285
```

Second, plot the first vector.

```
> plot(vg1, xlab="days of the week", ylab="occurrences of 'whatever'",  xaxt="n", type="b", main="")
```

In the plot call, add labels with `xlab` and `ylab`. `xaxt="n"` removes the automatic indexing along the *x*-axis. We need to do this to plot the names of the days instead. `type="b"` indicates the type of plot that we want: with **b**oth points and lines. The next line of code plots the names of the days. `grid()` adds a rectangular grid to the plot. With `lines()`, we plot the values of the second numeric vector.

```
> axis(1, at=1:7, labels=days)
> grid()
> lines(vg2, type="b", col="blue")
```

The mean and the standard deviation are calculated with their respective functions.

```
> #mean
> mean(vg1)
[1] 394.4286
> mean(vg2)
[1] 338.4286
> #standard deviation
> sd(vg1)
[1] 151.0572
> sd(vg2)
[1] 39.16145
```

We add the means to the plot with `abline()`.

```
abline(h=mean(vg1))
abline(h=mean(vg2), col="blue")
```

Finally, we annotate the plot with `text()`. The first two arguments of this function position the text on the plot. We specify the *x*-axis and the *y*-axis coordinates.

```
> text(1.2,mean(vg1)+10,"mean Valley girl 1", cex=0.8)
> text(1.2,mean(vg2)+10,"mean Valley girl 2", col="blue", cex=0.8)
> text(2,11,"mean girl 1", cex=0.6)
> text(2,9,"mean girl 2", col="blue", cex=0.6)
> text(5, 560, "Valley girl 1")
> text(5, 300, "Valley girl 2", col="blue")
```

Summarizing the data by means of a boxplot is equally easy.

```
> boxplot(vg1, vg2, names=c("Valley girl 1", "Valley girl 2"),
+         ylab="occurrences of 'whatever'")
```

Although both Valley girls use ‘whatever’ a lot, as evidenced by similar means, we observe much more deviation with respect to Valley girl 1.

Exercises of Chap. 8

8.1 Clear R's memory, load `whitman.txt`, and inspect.

```
> rm(list=ls(all=TRUE))
> whitman <- scan("/CLSR/chap4/whitman.txt", what="char", sep="\n")
> head(whitman,2)
[1] "When I heard the learn'd astronomer;"
[2] "When the proofs, the figures, were ranged in columns before me;"
```

The strategy that I adopt here is to look for letters, replace consonants with "C", vowels with "V", and count them. There are many ways of solving this exercise. The one I present here may not be the shortest, but it is simple to grasp.

First, we remove the apostrophes (as in *learn'd*) because we do not want it to interfere with the count.

```
> whitman <- gsub("'", "", whitman, ignore.case = TRUE)
```

Next, we split the text at non-word characters. This has the double advantage of getting rid of spaces and punctuation.

```
> whitman.split<- unlist(strsplit(whitman, "\\W+"))
> head(whitman.split)
[1] "When"       "I"          "heard"      "the"        "learnd"     "astronomer"
```

We paste the text back together so that we end up with an uninterrupted character string.

```
> whitman.paste <- paste(whitman.split, collapse ="")
```

With `gsub()`, we replace consonants (i.e. letters that are not "a" or "e" or "i" or "o" or "u") with "C" and vowels with "V".

```
> whitman <- gsub("[^aeiou]", "C", whitman.paste, ignore.case = TRUE)
> whitman <- gsub("[aeiou]", "V", whitman, ignore.case = TRUE)
```

Before counting the number of consonants and vowels, we need to find them first. We use `strapply()`, from the `gsubfn` package. The count proper is done with `length()`.

```
> library(gsubfn)
> nwhit <- sum(nchar(whitman)) # number of letters
> nC <- length(unlist(strapply(whitman, "C"))) # number of consonants
> nV <- length(unlist(strapply(whitman, "V"))) # number of vowels
```

The probability of drawing a consonant is `nC` divided by the total number of letters.

```
> P_C <-  nC/nwhit
> P_C
[1] 0.6256684
```

Because drawing a consonant and drawing a vowel are mutually exclusive, the probability of drawing a vowel is $1 - P(C)$.

```
> P_V <- 1-P_C
> P_V
[1] 0.3743316
```

The probability of drawing a consonant is 0.63 and the probability of drawing a vowel is 0.37.

8.2 Clear R's memory and load the contingency table. Convert the data frame into a matrix.

```
> rm(list=ls(all=TRUE))
> ct <- as.matrix(readRDS("C:/CLSR/chap8/ct.rds")) # Windows
> ct <- as.matrix(readRDS("/CLSR/chap8/ct.rds")) # Mac
```

Calculate the marginal totals (i.e. the sums of rows and columns) and add them to the matrix with `rbind()` and `cbind()`.

```
> row.totals <- colSums(ct)
> ct <- rbind(ct, row.totals)
> col.totals <- rowSums(ct)
> ct <- cbind(ct, col.totals)
> ct
            preadjectival predeterminer col.totals
quite                 453          1749       2202
rather               1423           370       1793
row.totals           1876          2119       3995
```

The cell at the intersection of `row.totals` and `col.totals` (`3995`) corresponds to the sample size. It can be obtained in two other ways.

```
> sum(rowSums(ct[1:2, 1:2]))
[1] 3995
> sum(colSums(ct[1:2, 1:2]))
[1] 3995
```

a. If we randomly selected an observation from this sample, the probability that this observation is $P(quite)$ is found by dividing the total frequency of *quite* by the sample size: $2202/3995 = 0.55$. With R, you do not need to calculate each probability individually. All you need to do is divide the matrix by the sample size. Let us round the results to two decimal places.

```
> ct_probs <- round(ct/ct[3,3], 2)
> ct_probs
            preadjectival predeterminer col.totals
quite                0.11          0.44       0.55
rather               0.36          0.09       0.45
row.totals           0.47          0.53       1.00
```

- $P(quite) = 0.55$ (55%)
- $P(rather) = 0.45$ (45%)
- $P(preadjectival) = 0.47$ (47%)
- $P(predeterminer) = 0.53$ (53%)

b. Calculating the probability of drawing the adverb *rather* in predeterminer position means finding the *intersection* of these two events (Sect. 8.2.4). This union is expressed in the form: $P(rather \cap predeterminer)$. It corresponds to the value in the table where *rather* and *predeterminer* intersect.

```
> P_predet_and_rather <- ct_probs[2,2]
> P_predet_and_rather
[1] 0.09
```

$P(rather \cap predeterminer) = 0.09$ (9%).

c. Calculating the probability of drawing *either* an adverb in predeterminer position *or* the adverb *rather* means finding the *union* of these two events (Sect. 8.2.4). This union is expressed in the form: $P(predeterminer \cup rather)$.

```
> P_predet <- ct_probs[3,2]
> P_rather <- ct_probs[2,3]
> P_predet_and_rather <- ct_probs[2,2]
> P_predet_or_rather <- P_predet + P_rather - P_predet_and_rather
> P_predet_or_rather
[1] 0.89
```

$P(predeterminer \cup rather) = 0.89$ (89%).

d. Let us first focus on the probability of an observation being in predeterminer position on the condition of being the adverb *rather*. We denote this as $P(predeterminer|rather)$. We calculate it by dividing the joint probability by the corresponding marginal probability for the *rather* sample only:

$$P(predeterminer|rather) = \frac{0.09}{0.45}.$$

```
> P_predet_cond_rather <- ct_probs[2,2]/ct_probs[2,3]
> P_predet_cond_rather
[1] 0.2
```

Given that an observation is the adverb *rather*, there is a 20% chance that it occurs in predeterminer position.

Now, let us do it the other way around. We calculate the probability for the observation to be the adverb *rather* on the condition of occurring in predeterminer position: $P(rather|predeterminer)$. We calculate it by dividing the joint probability by the corresponding marginal probability for the *predeterminer* sample only:

$$P(rather|predeterminer) = \frac{0.09}{0.53}.$$

```
> P_rather_cond_predet <- ct_probs[2,2]/ct_probs[3,2]
> P_rather_cond_predet
[1] 0.1698113
```

Given that an observation is in predeterminer position, there is a 17% chance that it is the adverb *rather*. Because the marginal probabilities are different, so are the conditional probabilities.

e. $P(preadjectival|quite)$ (i.e. the probability of drawing an observation in preadjectival position given that it is the adverb *quite*) is found by dividing the joint probability of the two events by the marginal probability of *quite*.

```
> P_preadj_cond_quite <- ct_probs[1,1]/ct_probs[1,3]
> P_preadj_cond_quite
[1] 0.2
```

$P(preadjectival|quite) = 0.2$ (20%).

$P(predeterminer|quite)$ (i.e. the probability of drawing an observation in predeterminer position given that it is the adverb *quite*) is found by dividing the joint probability of the two events by the marginal probability of *quite*.

```
> P_predet_cond_quite <- ct_probs[1,2]/ct_probs[1,3]
> P_predet_cond_quite
[1] 0.8
```

$P(predeterminer|quite) = 0.8$ (80%). We find that the adverb *quite* is much more likely to occur in predeterminer position in the corpus.

8.3 Clear R's memory, load the text, and inspect. Note that I have cleaned the text a little.

```
> rm(list=ls(all=TRUE))
> text <- scan("C:/CLSR/chap8/rodogune.txt", what="char", sep="\n") # Windows
> text <- scan("/CLSR/chap8/rodogune.txt", what="char", sep="\n") # Mac
> head(text)
```

Your first task is to determine the number of words per line. Use a loop.

```
> num_words <- integer() # create an empty integer vector to collect all results
> for (i in 1:length(text)){ # for each line in the text...
+   split_line <- unlist(strsplit(text[i], "\\W+")) # split at non-word characters...
+   num_words_line <- length(split_line)  # obtain the size in words of the line
+   num_words <- c(num_words, num_words_line) # collect all results
+ }
```

You are now ready to plot the histogram with `hist()`. You can add a density curve with `curve()` and `dnorm()`, as shown in the next line of code (Fig. B.4).

```
> hist(num_words, prob=TRUE, main="")
> curve(dnorm(x, mean=mean(num_words), sd=sd(num_words)), col="blue", add=TRUE)
```

According to the density curve, the data look normally distributed. However, when we look at the bars, we observe a little skew to the left. A Shapiro-Wilk normality test is needed to remove any doubt regarding the normality of the data. Remember that this test assumes the following:

H_0: the data are normally distributed;
H_1: the data are not normally distributed.

Thus, if $p < \alpha$, then H_0 is rejected because there is evidence that the data are not from a normally distributed population.

```
> shapiro.test(num_words)

Shapiro-Wilk normality test

data:  num_words
W = 0.9609, p-value < 2.2e-16
```

Here, $p < 2.2e{-}16$ (i.e. very close to zero). Despite what the histogram made us believe, the data are not normally distributed.

When you test for normality, make sure that your sample is large enough. Fig. B.5 shows the distribution of the data depending on the sample size. The larger the sample, the easier it is to make an opinion about the distribution.

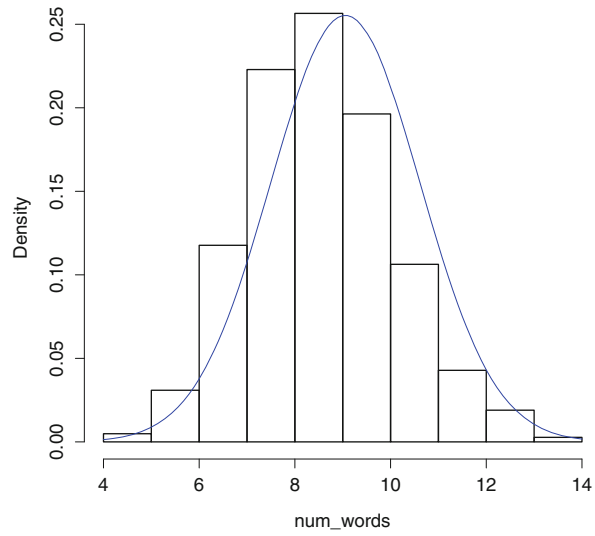


Fig. B.4: A histogram of words per lines in *Rodogune*

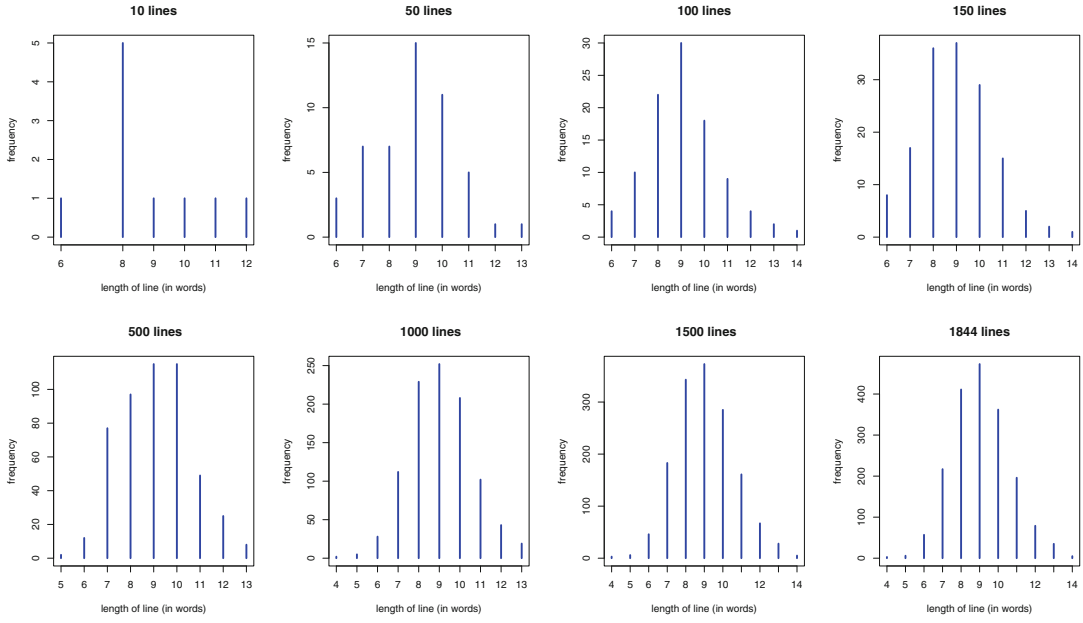


Fig. B.5: The distribution of words per lines in *Rodogune* based on different samples

8.4 First, we create the contingency table in the form of a matrix.

```
rm(list=ls(all=TRUE))
r1 <- c(353, 124)
r2 <- c(157, 42)
matrix <- matrix(c(r1,r2), nrow=2, ncol=2, byrow=TRUE); matrix
```

Next, we run a χ^2 test on the matrix and display the expected frequency of each cell.

```
chisq.test <- chisq.test(matrix)
round(chisq.test$expected, 0)
```

Rather than compare the observed frequencies to the expected frequencies, we access the `res` element from the `chisq.test()` output.

```
chisq.test$res
```

If we were to rely on observed frequencies only, we would naturally conclude that word 1 and word 2 tend to occur in context 1. With χ^2, we know that, in fact, word 1 disprefers context 1 and word 2 disprefers context 2. Conversely, the preference of word 1 for context 2 and the preference of word 2 for context 1 are significant.

8.5 First, we load the contingency table.

```
rm(list=ls(all=TRUE))
stance <- readRDS("/CLSR/chap8/stance.rds")
str(stance)
```

First, we need to convert the counts in each column into ordinal data. This can be done with `rank()`.

```
for (i in 1:ncol(stance)){
  stance[,i] <- rank(stance[,i])
}
```

With `cor()` and the `method="kendall"` argument, we can now compute the entire matrix of correlations between all the variables in `stance`.

```
cor(stance, method="kendall")
```

With `cor.test()` and the `method="kendall"` argument, we calculate the pairwise correlations.

```
cor.test(stance$ACPROSE, stance$NEWS, method="kendall")
```

There is a strong positive correlation between the word distribution of adjective categories in `ACPROSE` and `NEWS` ($\tau = 0.73$). The correlation is slightly above 0.05 ($p = 0.06$).

```
cor.test(stance$ACPROSE, stance$NONAC, method="kendall")
```

There is a perfect positive correlation between the word distribution of adjective categories in `ACPROSE` and `NONAC` ($\tau = 1$). The correlation is quite significant ($p = 0.002$).

```
cor.test(stance$NEWS, stance$CONVRSN, method="kendall")
```

There is a weak negative correlation between the word distribution of adjective categories in `NEWS` and `CONVRSN` ($\tau = -0.11$), but the correlation is not significant ($p = 0.77$). Mind you: because there are ties, the exact p-value is not computed.

Exercise of Chap. 9

The simple, non-interactive function requires that you vectorize the frequency of the target word (`freq_W1`) and the corpus size in word tokens (`corp.size`).

```
> rm(list=ls(all=TRUE))
> freq_W1 <- 63135
> corp.size <- 533788932
```

You should also load the input file beforehand.

```
> data <- readRDS("C:/CLSR/chap9/computer.NP.rds") # Windows
> data <- readRDS("/CLSR/chap9/computer.NP.rds") # Mac
```

The vector `data` is the sole argument of your function. This unique argument is declared as `x` in the structure of the function.

```
> assoc.chi <- function(x, ...){
+
+ }
```

Just place the loop inside the function. Optionally, you may append a line of code right after the loop prompting the user to choose an output file interactively. This file is where the results are stored.

```
> assoc.chi <- function(x, ...){
+ all.chi <- character()
+ for (i in 1:nrow(data)){ # enter the loop
+  a <- data$freq_W1_W2[i] # cell a
+  b <- freq_W1 - a # cell b
+  c <- data$corp_freq_W2[i] - a # cell c
+  d <- (corp.size - freq_W1) - c # cell d
+  m <- matrix(c(a, b, c, d), nrow=2, ncol=2, byrow=TRUE) # make the 2x2 matrix
+  N <- sum(m) # sum total of the contingency table
+  N_row1 <- rowSums(m)[1] # sum total of row 1
+  N_row2 <- rowSums(m)[2] # sum total of row 2
+  N_col1 <- colSums(m)[1] # sum total of column 1
+  N_col2 <- colSums(m)[2] # sum total of column 2
+  E11 <- (N_row1*N_col1/N) # expected freq of cell in row 1 col 1
+  E12 <- (N_row1*N_col2/N) # expected freq of cell in row 1 col 2
+  E21 <- (N_row2*N_col1/N) # expected freq of cell in row 2 col 1
+  E22 <- (N_row2*N_col2/N) # expected freq of cell in row 2 col 2
+  m.exp <- matrix(c(E11, E12, E21, E22), nrow=2, ncol=2, byrow=TRUE) # matrix of expected freqs
+  current.chi <- ((m[1,1]-m.exp[1,1])^2/m.exp[1,1]) +
+    ((m[1,2]-m.exp[1,2])^2/m.exp[1,2]) +
+    ((m[2,1]-m.exp[2,1])^2/m.exp[2,1]) +
+    ((m[2,2]-m.exp[2,2])^2/m.exp[2,2])
+   current.chi <- round(current.chi, 2) # round the score
+   current.chi <- paste(data$W2[i], current.chi, sep="\t") # append word 2
+  all.chi <- c(all.chi, current.chi) # collect all scores
+ } # exit the loop
+ cat(all.chi, file=file.choose(), sep="\n")
+ }
```

To run the function, enter `assoc.chi()` and choose `data` as its unique argument.

```
> assoc.chi(data)
```

An interactive window opens almost immediately. Select the output text file. You are done.

With an interactive function, you do not have to worry about entering data before running the script. Because the function has no argument, leave the brackets of `function()` empty. The first part of the function prompts the user to select a format for the input file. The `menu()` function asks a question formulated in the `title` argument. The question offers two options: `"R data format (.rds)"` and `"tab-separated text file (.txt)"`. The `switch()` function tells R what to do based on the user's choice.

```
> assoc.chi <- function(){ # beginning of the function
+ what.format <- switch(menu(c("R data format (.rds)", "tab-separated text file (.txt)"),
+                       title="\nWhat is the format of the input file? (enter 0 to exit)"),
+                       load.data.rds(), load.data.txt())
+ } # end of the function
```

If the data is an R data file (choice: `1`), the script proceeds to launch `load.data.rds()`, and if the data is a tab-delimited text file (choice: `2`), the script proceeds to launch `load.data.txt()`. These subfunctions do two things:

- they load the input file using the functions required for each kind (`readRDS()` for a R data file, `read.table()` for a tab-delimited text file);
- they launch another function: `assoc.chi.2()`.

```
> load.data.rds <- function(){
+ readline("Press Enter to select the input matrix (.rds)...")
+ data <- readRDS(file=file.choose())
+ assoc.chi.2()
+ }
>
> load.data.txt <- function(){
+ readline("Press Enter to select the input matrix (.txt)...")
+ data <- read.table(file=file.choose(), header=T, sep="\t")
+ assoc.chi.2()
+ }
```

The `assoc.chi.2()` function is the most important part of the script. It is an interactive version of the loop in Sect. 9.3.4.

```
> assoc.chi.2 <- function(){
+ x <- readline("What is the frequency of W1, a.k.a. the target word?\n")
+ freq_W1 <- as.numeric(x)
+ y <- readline("what is the corpus size in word tokens?\n")
+ corp.size <- as.numeric(y)
+ all.chi <- character()
+ for (i in 1:nrow(data)){ # enter the loop
+  a <- data$freq_W1_W2[i] # cell a
+  b <- freq_W1 - a # cell b
+  c <- data$corp_freq_W2[i] - a # cell c
+  d <- (corp.size - freq_W1) - c # cell d
+  m <- matrix(c(a, b, c, d), nrow=2, ncol=2, byrow=TRUE) # make the 2x2 matrix
+  N <- sum(m) # sum total of the contingency table
+  N_row1 <- rowSums(m)[1] # sum total of row 1
+  N_row2 <- rowSums(m)[2] # sum total of row 2
+  N_col1 <- colSums(m)[1] # sum total of column 1
+  N_col2 <- colSums(m)[2] # sum total of column 2
+  E11 <- (N_row1*N_col1/N) # expected freq of cell in row 1 col 1
+  E12 <- (N_row1*N_col2/N) # expected freq of cell in row 1 col 2
+  E21 <- (N_row2*N_col1/N) # expected freq of cell in row 2 col 1
+  E22 <- (N_row2*N_col2/N) # expected freq of cell in row 2 col 2
+  m.exp <- matrix(c(E11, E12, E21, E22), nrow=2, ncol=2, byrow=TRUE) # matrix of expected freqs
+  current.chi <- ((m[1,1]-m.exp[1,1])^2/m.exp[1,1]) +
+    ((m[1,2]-m.exp[1,2])^2/m.exp[1,2]) +
+    ((m[2,1]-m.exp[2,1])^2/m.exp[2,1]) +
+    ((m[2,2]-m.exp[2,2])^2/m.exp[2,2])
+   current.chi <- round(current.chi, 2) # round the score
+   current.chi <- paste(data$W2[i], current.chi, sep="\t") # append word 2
+  all.chi <- c(all.chi, current.chi) # collect all scores
+ } # exit the loop
+ readline("Press Enter to select a text file where you want to save the results...")
+ output <- file.choose() # select the output file
+ cat(all.chi, file=output, sep="\n") # print the results into the output file
+ cat(paste("Here is the path to your output file: ",
+           normalizePath(output), sep="")) # print the path
```

```
+ }
```

First, the function prompts the user to input the frequency of the target word and the corpus size in word tokens. Next comes the loop. Upon exiting the loop, the function prompts the user to select a text file where the results are saved. The selected file is vectorized in `output`, so that its path can be retrieved after the results have been saved.

The full function is stored in `(C:)/CLSR/chap9`. To load it, use `source()`. To run it, enter `assoc.chi()`.

```
> source("/CLSR/chap9/assoc.chi.R") # Windows
> source("/CLSR/chap9/assoc.chi.R") # Mac
> assoc.chi()
```

Exercises of Chap. 10

10.1 Clear R's memory and load the data frame

```
> rm(list=ls(all=TRUE))
> df <- readRDS("C:/CLSR/chap10/it.be.A.to.V.that.rds") # Windows
> df <- readRDS("/CLSR/chap10/it.be.A.to.V.that.rds") # Mac
```

Inspect the data frame.

```
> str(df)
'data.frame': 244 obs. of  7 variables:
 $ category         : Factor w/ 2 levels "ADJ","V_inf": 1 1 2 1 2 1 2 2 2 1 ...
 $ num.hapax        : int  1 5 8 1 1 0 2 0 1 2 ...
 $ V                : int  1 10 15 1 1 2 4 2 3 2 ...
 $ delta.diff       : num  -6.09e-05 1.13e-03 3.61e-04 1.70e-04 -1.52e-03 ...
 $ association.chisq: num  1.54 99.67 6.17 1.26 5.2 ...
 $ P.potential      : num  0.00051 0.00255 0.00408 0.00051 0.00051 ...
 $ P.global         : int  1960 3920 3675 1960 1960 999999999 3920 999999999 1960 1960 ...
> head(df)
           category num.hapax  V    delta.diff association.chisq  P.potential  P.global
able            ADJ         1  1 -6.091308e-05          1.540000 0.0005102041      1960
absurd          ADJ         5 10  1.125578e-03         99.671429 0.0025510204      3920
accept        V_inf         8 15  3.613657e-04          6.167273 0.0040816327      3675
acceptable      ADJ         1  1  1.699959e-04          1.260000 0.0005102041      1960
access        V_inf         1  1 -1.520983e-03          5.200000 0.0005102041      1960
accurate        ADJ         0  2  6.467982e-04          0.560000 0.0000000000 999999999
```

Next, we load the `FactoMineR` package.

```
> library(FactoMineR)
```

We perform the PCA with the `PCA()` function. We indicate that the first column is a qualitative variable with `quali=1`. We also indicate that the second and third columns are quantitative and supplementary: `quanti.sup=c(2,3)`. We specify `graph=FALSE` as we want to choose our own plotting parameters.[3]

```
> pca.object <- PCA(df, quali=1, quanti.sup=c(2,3), graph=F)
```

At this stage, it is advisable to inspect the eigenvalues associated with each component (or dimension). Eigenvalues measure the relative importance of the components. Their ranking, based on their respective contributions, helps us select the components that we should take into account.

[3] Enter `?plot.PCA` for details.

```
> round(pca.object$eig, 2)
       eigenvalue percentage of variance cumulative percentage of variance
comp 1       1.14                 28.60                            28.60
comp 2       1.13                 28.37                            56.97
comp 3       0.89                 22.31                            79.28
comp 4       0.83                 20.72                           100.00
```

The above table of contributions is summarized graphically by the barplot in Fig. B.6.

```
> barplot(pca.object$eig[,2], names=paste("dim", 1:nrow(pca.object$eig)),
+         xlab="dimensions", ylab="percentage of variance")
```

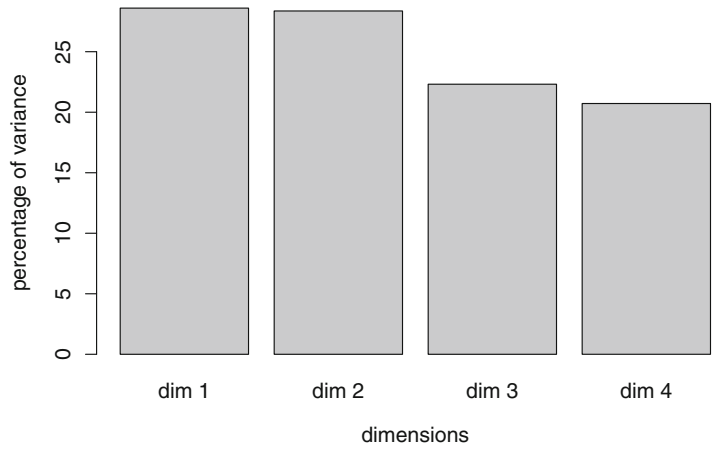


Fig. B.6: A barplot showing the eigenvalues associated with each dimension

After the second dimension, we observe a decrease of eigenvalues, but this decrease is not sharp enough. In fact, if we focused on the first two dimensions, we would be missing $100 - 56.97 = 43.03\%$ of the variance. In this case, we shall have to take all four dimensions into account.

First, we plot the circle of correlations between variables and the graph of individuals for the first two components (Fig. B.7). As you know, the former is a key for the interpretation of the latter.

```
> par(mfrow=c(2,1))
> plot(res.pca, cex=0.9,shadow=T, autoLab="yes", choix="var", title="")
> plot(res.pca, cex=0.9, shadow=T, autoLab="yes", palette=palette(c("black","blue")),
+      habillage=1, title="")
```

If we examine the circle of correlations, we see a perfect correlation between `P.potential` and `num.hapax` with respect to the first two components. This is normal because the former is based on the latter. We would also expect a strong positive correlation between `P.global` and `V` because the former is also based on the latter, but such is not the case. This correlation is perhaps visible if we inspect other components. The variables `association.chisq` and `delta.diff` seem to be positively correlated with respect to the first two components, which is counterintuitive and somewhat contradictory. As we shall see, their relation is better captured by the third and fourth components. The most salient conclusion that we draw

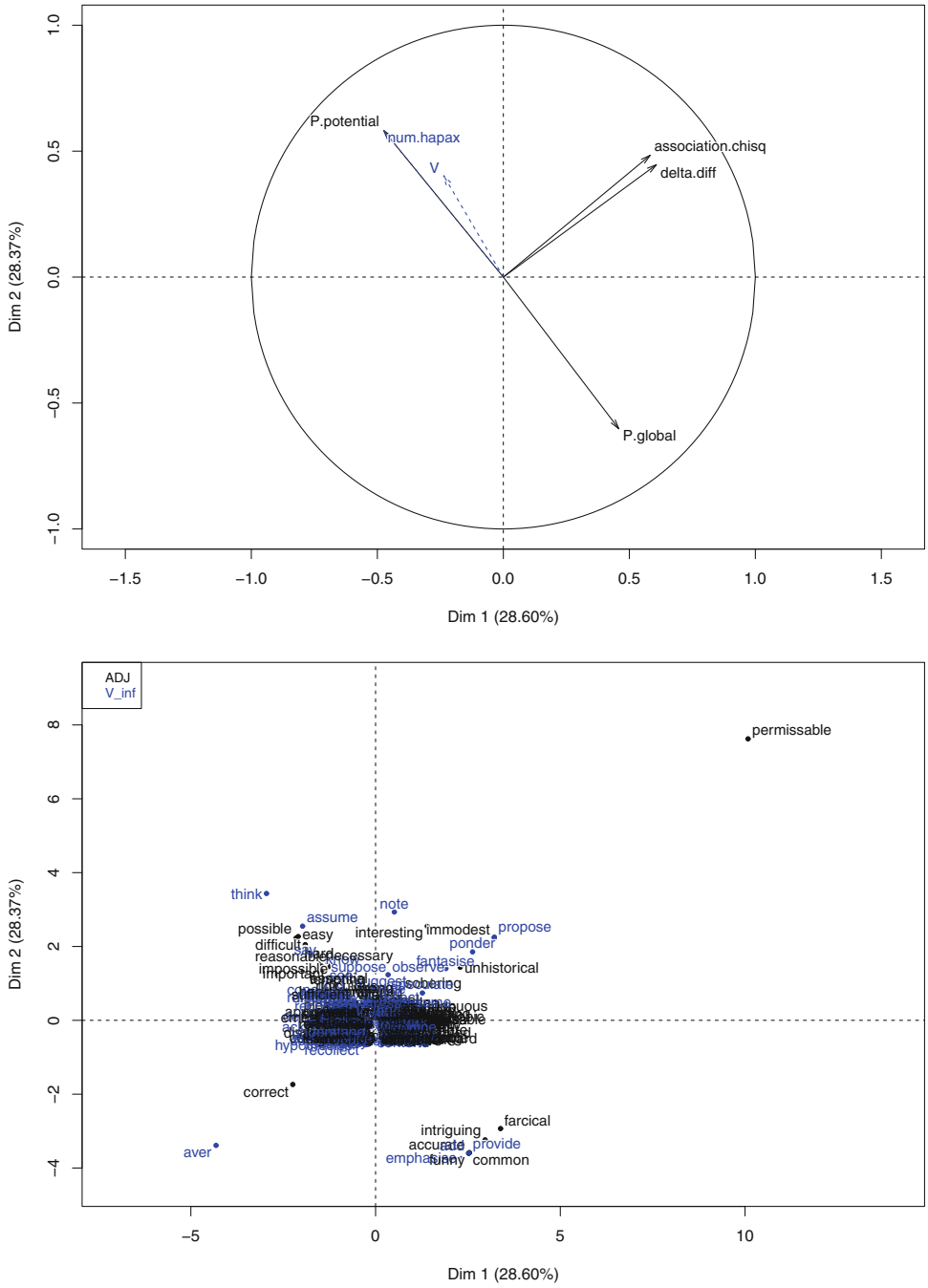


Fig. B.7: PCA biplots of adjectives and infinitive verbs in *it BE ADJ to V that* in the BNC—components 1 and 2

from the inspection of the first two components is that the variables `P.potential` and `P.global` are opposed. Remember that potential productivity ($\mathscr{P}$) measures the probability of encountering new types, whereas global productivity (P^*) evaluates $\mathscr{P}$ in terms of observed types. If we examine the graph of individuals, the probability of encountering new types is higher for those subschemas in the upper left corner: *it BE ADJ to think/assume/say* and *it BE possible/easy/difficult/hard/reasonable/impossible/necessary to V_inf*. The global productivity is higher for those subschemas in the lower right corner: *it BE ADJ to add/provide/emphasise* and *it BE intriguing/farcical/accutate/funny/common to V_inf*. The subschema *it BE permissable to V_inf* in the upper right corner of the plot is characterized by a high score of either symmetric association or ΔP difference. A quick look at our data table shows that the subschema is in fact characterized by a high symmetric-association score.

```
> df[which(row.names(df)=="permissable"),]
            category num.hapax V delta.diff association.chisq  P.potential P.global
permissable      ADJ         1 1  0.1658178          13720.11 0.0005102041     1960
```

The position of *it BE correct that V_inf* and *it is ADJ to aver that* in the lower left corner is explained by their negative ΔP differences.

```
> df[which(row.names(df)=="aver"),]
     category num.hapax V delta.diff association.chisq  P.potential P.global
aver    V_inf         1 1 -0.1998271           3351.24 0.0005102041     1960
> df[which(row.names(df)=="correct"),]
        category num.hapax V delta.diff association.chisq P.potential P.global
correct      ADJ         2 3  -0.100011          1677.785 0.001020408     1960
```

Next, we plot the circle of correlations between variables and the graph of individuals for the third and fourth components (Fig. B.8).

```
> par(mfrow=c(2,1))
> plot(res.pca, axes=c(3,4), cex=0.9,shadow=T, autoLab="yes", choix="var", title="")
> plot(res.pca, axes=c(3,4), cex=0.9, shadow=T, autoLab="yes", palette=palette(c("black","blue")),
+      habillage=1, title="")
```

This time, the most conspicuous opposition is between `association.chisq` (upper left corner of the correlation circle) and `delta.diff` (lower right corner). We were right to wait until we inspect components 3 and 4 before getting to a conclusion. The subschemas that cluster in the upper left corner of individuals are characterized by a high symmetric-association score: *it is ADJ to aver/fantasise/hypothesise/note/ponder that* and *it is correct/inresting/logical to V_inf that*. These subschemas appear in somewhat fixed ADJ–V_inf combinations. The subschemas that cluster in the upper right corner of the graph of individuals are the most potentially and globally productive: *it is ADJ to think/emphasise/provide/add/assume/know/suppose/say that* and *it is intriguing/accurate/common/funny/farcical/easy/possible/difficult/hard/important/reasonable to V_inf that*. They appear in more flexible ADJ–V_inf combinations.

10.2 First, clear R's memory and load the data frame.

```
> rm(list=ls(all=TRUE))
> df <- readRDS("C:/CLSR/chap10/modals.BNC.rds") # Windows
> df <- readRDS("/CLSR/chap10/modals.BNC.rds") # Mac
```

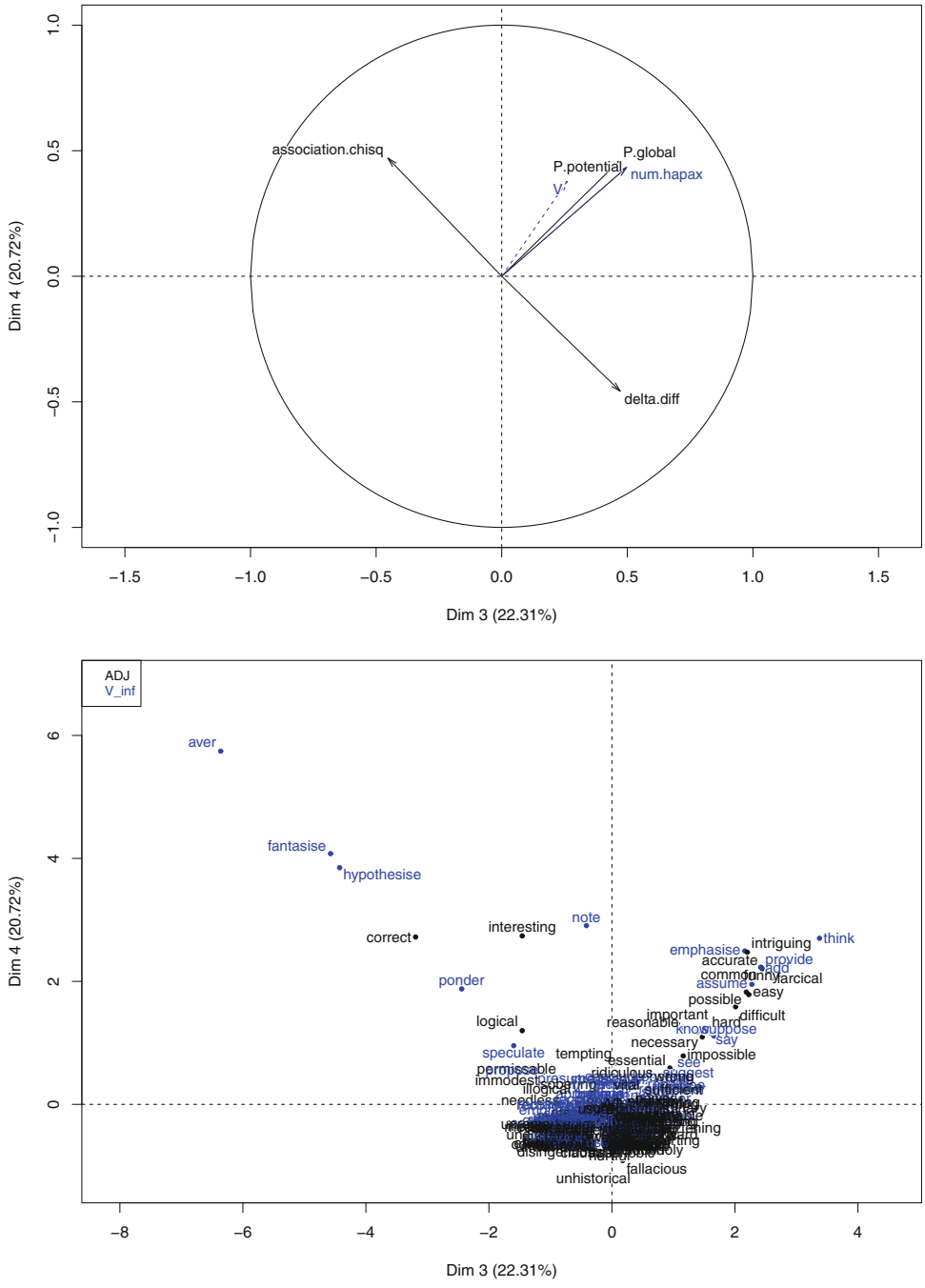


Fig. B.8: PCA biplots of adjectives and infinitive verbs in *it BE ADJ to V that* in the BNC—components 3 and 4

Because the data frame is very wide, we inspect only a portion of it.

```
> df[, sample(1:ncol(df), 5)]
       S_interview_oral_history S_pub_debate W_ac_humanities_arts S_brdcast_documentary
can                        1832         1048                8251                   126
could                      2208          557                5103                    68
may                         141          313                5190                    35
might                       283          191                3123                    22
must                        348          127                2517                     8
ought                        31           63                 219                     3
shall                        68           51                 654                     2
should                      359          650                2863                    40
will                        379          817                3820                    97
would                      3695         2130                6868                   142
       S_lect_humanities_arts
can                       214
could                     135
may                        39
might                      31
must                       43
ought                      12
shall                      17
should                     33
will                      117
would                     200
```

With `grep()`, we isolate the variables that correspond to written and spoken texts. This will come in handy later on when we plot each kind of variable separately.

```
> written.vars <- grep("W_", colnames(df), value=TRUE)
> head(written.vars)
[1] "W_ac_humanities_arts" "W_ac_medicine"        "W_ac_nat_science"     "W_ac_polit_law_edu"
[5] "W_ac_soc_science"     "W_ac_tech_engin"
> spoken.vars <- grep("S_", colnames(df), value=TRUE)
> head(spoken.vars)
[1] "S_brdcast_discussn"    "S_brdcast_documentary" "S_brdcast_news"        "S_classroom"
[5] "S_consult"             "S_conv"
```

Next, we load the `FactoMineR` package.

```
> library(FactoMineR)
```

We perform the correspondence analysis with the `CA()` function. We specify `graph=FALSE` as we want to choose our own plotting parameters.[4]

```
> ca.object <- CA(df, graph=FALSE)
```

[4] Enter `?plot.CA` for details.

At this stage, it is advisable to inspect the eigenvalues associated with each dimension.

```
> round(ca.object$eig, 2)
      eigenvalue percentage of variance cumulative percentage of variance
dim 1       0.09                 51.26                             51.26
dim 2       0.04                 22.92                             74.18
dim 3       0.03                 13.93                             88.11
dim 4       0.01                  4.53                             92.64
dim 5       0.01                  4.15                             96.79
dim 6       0.00                  1.33                             98.12
dim 7       0.00                  1.05                             99.17
dim 8       0.00                  0.56                             99.73
dim 9       0.00                  0.27                            100.00
```

The above table of contributions is summarized graphically by the barplot in Fig. B.9.

```
> barplot(ca.object$eig[,2], names=paste("dim", 1:nrow(ca.object$eig)),
+         xlab="dimensions", ylab="percentage of variance")
```

The first three dimensions represent 88.11% of all the variance in the data table. After the third dimension, we observe a sharp decrease of eigenvalues. This means that the contributions of dimensions 4, 5, 6, 7, 8, and 9 are not decisive. Therefore, we can safely ignore them.

Because we want to compare more than two dimensions, we use the `axes` option and make two CA plots on top of each other with `par(mfrow=c(1,2))`. The `selectCol` argument allows you to select the column variables that are plotted. We use it to select the variables that correspond to written texts (`written.vars`). The `unselect` argument can take two values: `0` or `1`. If `unselect=0`, the variables that are not selected (spoken texts in this case) are plotted without labels. If `unselect=1`, the unselected variables are not plotted at all. We choose the latter, so as not to clutter the graph (Fig. B.10).

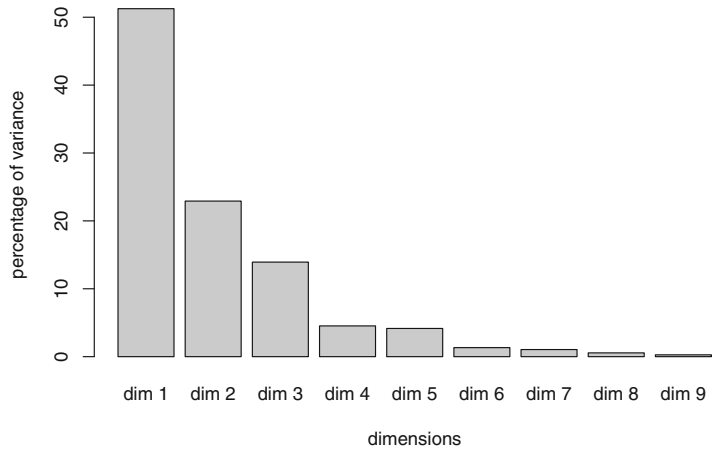


Fig. B.9: A barplot showing the eigenvalues associated with each dimension

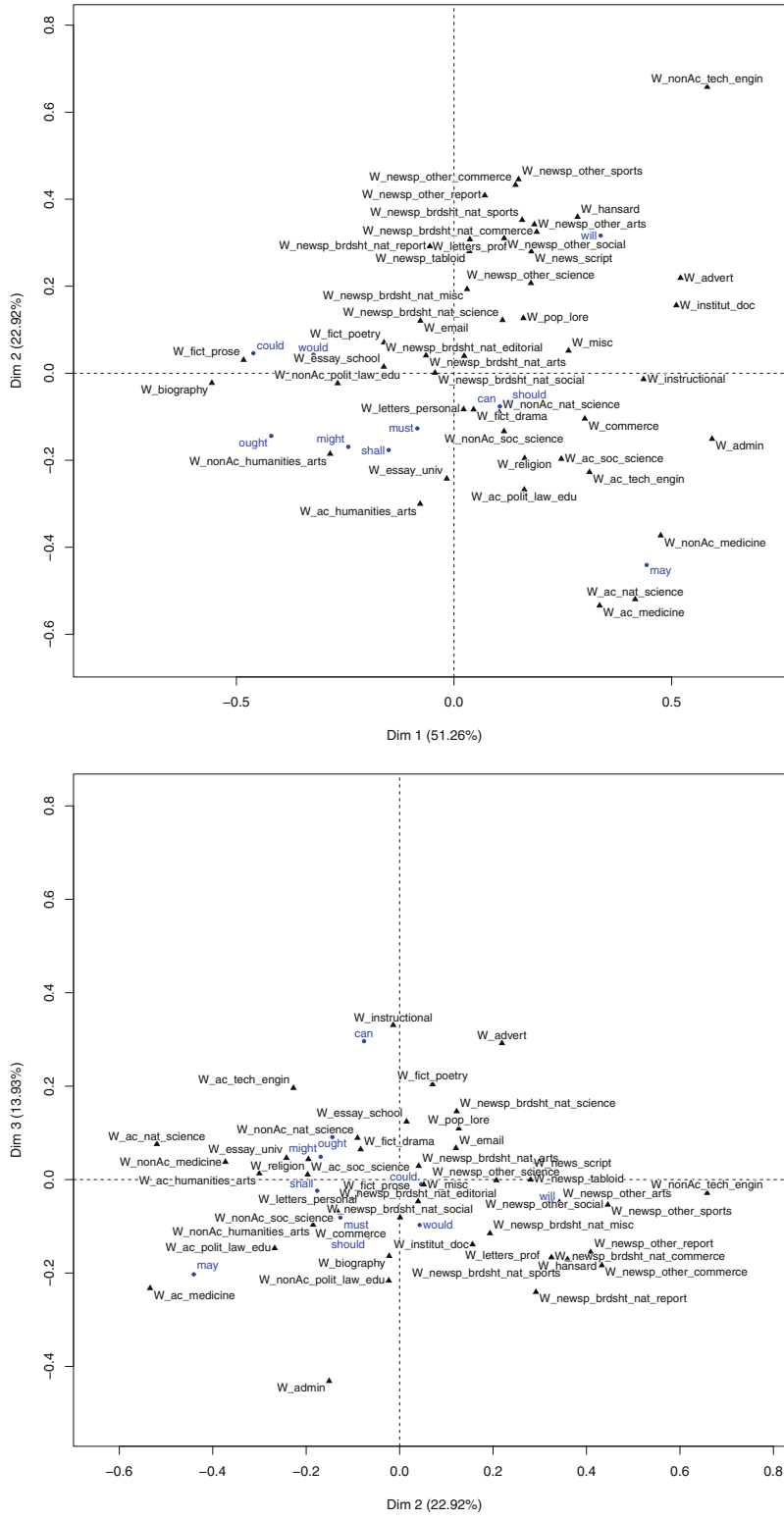


Fig. B.10: The distribution of 10 modals across 46 written-text genres. *Top*: dimensions 1 and 2; *bottom*: dimensions 2 and 3

```
> par(mfrow=c(1,2))
> # dimensions 1 and 2
> plot(ca.object, autoLab="yes", shadow=TRUE, col.col="black", col.row="blue",
+      selectCol = written.vars, unselect=1, title="", cex=0.8)
> # dimensions 2 and 3
> plot(ca.object, axes = c(2, 3), autoLab="yes", shadow=TRUE, col.col="black", col.row="blue",
+      selectCol = written.vars, unselect=1, title="", cex=0.8)
```

Inspection of dimensions 1 and 2 reveals that *may* is associated with scientific texts in the fields of medicine (both in academic and non-academic prose) and natural sciences. *Might* (together with *ought*) seems to be preferred in non-academic humanities. Inspection of dimensions 2 and 3 reveals confirms the preference of *may* for the medical context. Dimension 3 reveals that this modal is also used significantly in academic prose bearing on politics, law, and education. The same dimension confirms the proximity between *might* and *ought*.

The CA plot of the spoken variables is obtained in a similar way (Fig. B.11). All that we need to change is the `selectCol` argument and point it to `spoken.vars`.

```
> par(mfrow=c(1,2))
> # dimensions 1 and 2
> plot(ca.object, autoLab="yes", shadow=TRUE, col.col="black", col.row="blue",
+      selectCol = spoken.vars, unselect=1, title="", cex=0.8)
> # dimensions 2 and 3
> plot(ca.object, axes = c(2, 3), autoLab="yes", shadow=TRUE, col.col="black", col.row="blue",
+      selectCol = spoken.vars, unselect=1, title="", cex=0.8)
```

In both plots, the profile of *may* stands out the most. Inspection of dimensions 1 and 2 (top plot) reveals that *may* is preferred in lectures (`S_lect`). In the same plot, we see that *will* is preferred in spoken contexts where speech is scripted: TV or radio news broadcasts, planned speech, whether dialogue or monologue, parliamentary speeches, and religious sermons.

The above comments are based on a toy data set. Ideally, we should further analyze the use of these modals by determining if they are deontic or epistemic, or examining what verbs they occur with, for instance.

10.3 We clear R's memory and load the data.

```
> rm(list=ls(all=TRUE))
> df <- readRDS("C//CLSR/chap10/typocorp.rds") # Windows
> df <- readRDS("/CLSR/chap10/typocorp.rds") # Mac
```

The individuals (the corpora) have their own columns, as you can see when you inspect the data.

```
> str(df)
'data.frame': 17 obs. of  8 variables:
 $ corpus               : Factor w/ 17 levels "Bank of English",..: 12 14 16 7 17 1 8 11 5 6 ...
 $ variety              : Factor w/ 3 levels "var: GB","var: international",..: 1 1 1 1 3 2 2 2 3
    3 ...
 $ general_specific     : Factor w/ 2 levels "general","specific": 2 2 1 2 2 1 1 1 1 2 ...
 $ static_dynamic       : Factor w/ 2 levels "dynamic","static": 2 2 2 2 2 1 2 2 2 2 ...
 $ synchronic_diachronic: Factor w/ 2 levels "diachronic","synchronic": 1 1 2 2 2 2 2 2 2 1 ...
 $ stored_data_format   : Factor w/ 3 levels "text","text + audio",..: 1 1 1 1 2 1 1 1 1 1 ...
 $ mode                 : Factor w/ 3 levels "spoken","spoken + written",..: 3 3 1 1 1 2 3 3 2 3 ...
 $ size                 : Factor w/ 5 levels "large","largest",..: 3 3 5 5 5 2 4 4 1 1 ...
```

Next, we load the `FactoMineR` package.

```
> library(FactoMineR)
```

We perform the MCA with the `MCA()` function. We indicate that the first, sixth, and eighth columns are qualitative and supplementary: `quali.sup=c(1,6,8)`. We specify `graph=FALSE` as we want to choose our own plotting parameters.

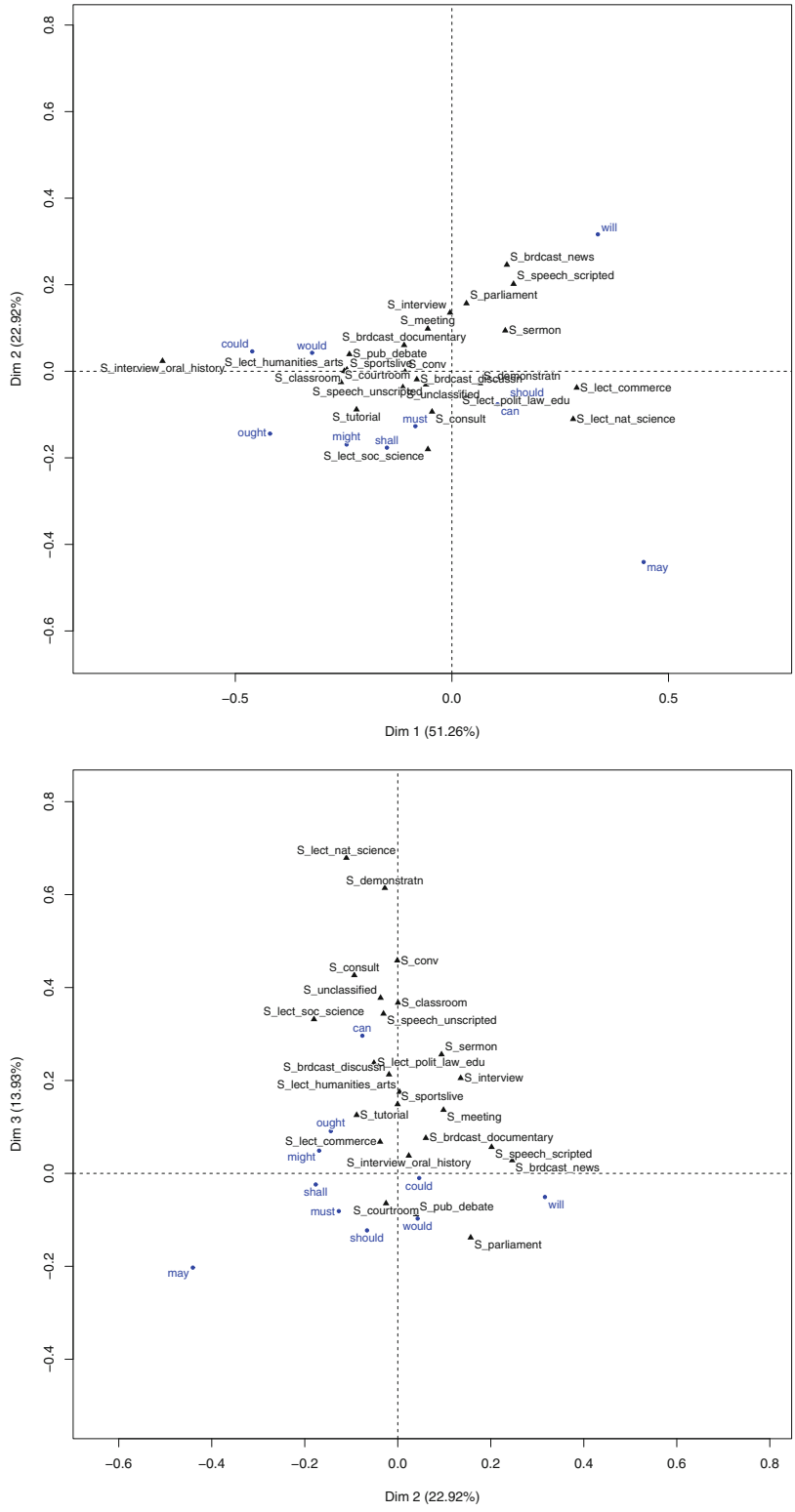


Fig. B.11: The distribution of 10 modals across 24 spoken-text genres. *Top*: dimensions 1 and 2; *bottom*: dimensions 2 and 3

```
> mca.object <- MCA(df, quali.sup=c(1,6,8), graph=FALSE)
```

We inspect the eigenvalues associated with each dimension. The numeric table below is graphically represented in Fig. B.12.

```
> round(mca.object$eig, 2)
       eigenvalue percentage of variance cumulative percentage of variance
dim 1        0.45                 32.08                            32.08
dim 2        0.30                 21.14                            53.22
dim 3        0.25                 17.72                            70.95
dim 4        0.20                 14.34                            85.29
dim 5        0.14                 10.32                            95.61
dim 6        0.04                  2.62                            98.23
dim 7        0.02                  1.77                           100.00
```

The expected sharp decrease occurs after dimension 5. Ideally, we should inspect five dimensions. In this exercise, however, we shall focus on the first two dimensions, which account for 53.22% of the variance. In MCA, it is not unusual to see the eigenvalue percentages drop dramatically when you demultiply the variables and their associated categories. This means more dimensions to inspect.

Before we plot the MCA graph, we should remember that the rows are indexed from 1 to 17. Because the corpora are declared in a separate column, the row names are in fact `"1"`, `"2"`, ..., `"17"`. As we do not want to plot these indices, we specify `invisible=c("ind")` so as to make the individuals (i.e. the row names) invisible. The argument `habillage="quali"` assigns one color per qualitative variable. You obtain the graph in Fig. 3.1.

```
> plot(mca.object, invisible=c("ind"), autoLab="auto", shadow=TRUE, habillage="quali", cex=.6, title="")
```

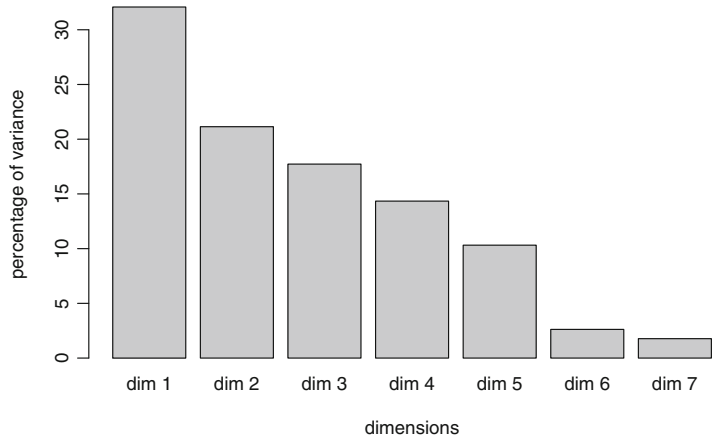


Fig. B.12: A barplot showing the eigenvalues associated with each dimension

10.4 As usual, we clear R's memory and load the data.

```
> rm(list=ls(all=TRUE))
> df <- readRDS("C//CLSR/chap10/modals.BNC.rds") # Windows
> df <- readRDS("/CLSR/chap10/modals.BNC.rds") # Mac
```

Create one distance object per distance metric.

```
> dist.object.euclidean <- dist(df, method="euclidean", diag=T, upper=T)
> dist.object.manhattan <- dist(df, method="manhattan", diag=T, upper=T)
> dist.object.minkowski <- dist(df, method="minkowski", diag=T, upper=T)
> dist.object.canberra <- dist(df, method="canberra", diag=T, upper=T)
```

Use `hclust()` to apply Ward's amalgamation rule on each distance object.

```
> clusters.euclidean <- hclust(dist.object.euclidean, method="ward.D")
> clusters.manhattan <- hclust(dist.object.manhattan, method="ward.D")
> clusters.minkowski <- hclust(dist.object.minkowski, method="ward.D")
> clusters.canberra <- hclust(dist.object.canberra, method="ward.D")
```

Finally, plot the four dendrograms (Fig. B.13).

```
> par(mfrow=c(2,2))
> plot(clusters.euclidean, sub="(Euclidean, Ward)", main="")
> plot(clusters.manhattan, sub="(Manhattan, Ward)", main="")
> plot(clusters.minkowski, sub="(Minkowski, Ward)", main="")
> plot(clusters.canberra, sub="(Canberra, Ward)", main="")
```

The dendrograms based on the Euclidean, Manhattan, and Minkowski metrics are almost identical. The scale of "Height" in the Manhattan dendrogram is different, but the shape of the tree is not affected. The Canberra method clusters the individuals differently. In my experience, the Canberra metric works best when the matrix is sparse, i.e. when there are many cells with null values.

10.5 Clear R's memory and load the `igraph` library, the vertex attributes, and the edge list.

```
> rm(list=ls(all=TRUE))
> library(igraph)
> v.attr <- readRDS("C:/CLSR/chap10/v.attr.practice.rds")  # Windows
> e.list <- readRDS("C:/CLSR/chap10/e.list.practice.rds") # Windows
> v.attr <- readRDS("/CLSR/chap10/v.attr.practice.rds")  # Mac
> e.list <- readRDS("/CLSR/chap10/e.list.practice.rds") # Mac
```

Select those vertices whose frequency is greater than or equal to 20 with `which()`.

```
> v.attr <- v.attr[which(v.attr$Const_freq >= 20),]
```

Now, select the items from the `from` and `to` columns of the edge list that are also present in the vertex attributes. You do it by subsetting `e.list` and using the `%in%` operator.

```
> e.list <- e.list[e.list$from %in% v.attr$Name,]
> e.list <- e.list[e.list$to %in% v.attr$Name,]
```

Inspect the data.

```
> head(v.attr)
        Name Type Const_freq
26 difficult    A        123
32      easy    A        132
35 essential    A         23
38      fair    A         96
45      good    A         46
48      hard    A        132
> head(e.list)
          from        to assoc.chisq
3  interesting      note   112572.26
```

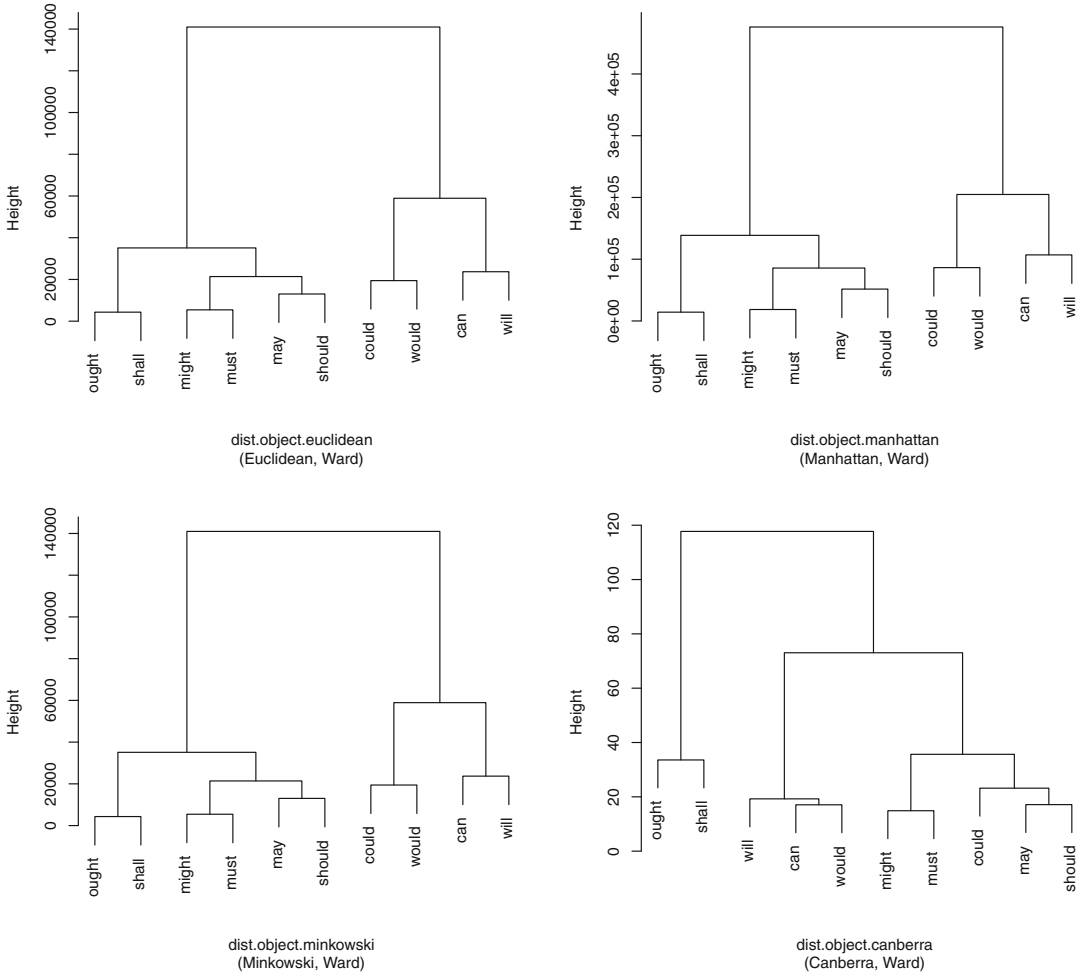


Fig. B.13: Cluster dendrograms of ten English modals in the BNC

```
5    important      note    15394.75
7    important    stress     2356.39
11   important   realize      649.61
13   important recognize      597.73
14   important   realise      975.63
```

It is now time to create the graph object with `graph.data.frame()`. Provide the edge list, the vertex attributes (`vertices=v.attr`), and specify that you want a directed graph (`directed="TRUE"`).

```
> G <- graph.data.frame(e.list, vertices=v.attr, directed="TRUE")
> G
IGRAPH DN-- 39 187 --
+ attr: name (v/c), Type (v/c), Const_freq (v/n), assoc.chisq (e/n)
+ edges (vertex names):
 [1] interesting->note      important  ->note      important  ->stress    important  ->realize
 [5] important  ->recognize important  ->realise   reasonable ->suppose   reasonable ->assume
```

```
 [9] important  ->remember  interesting->observe   important  ->recognise sufficient ->note
[13] possible   ->argue     interesting->note      easy       ->assume    difficult  ->argue
[17] easy       ->forget    difficult  ->imagine   safe       ->assume    hard       ->imagine
[21] important  ->observe   difficult  ->believe   important  ->ensure    interesting->note
[25] difficult  ->suppose   hard       ->believe   interesting->observe   important  ->stress
[29] essential  ->stress    essential  ->stress    sufficient ->stress    important  ->recognize
+ ... omitted several edges
```

Adjust the vertex size using the construction frequency. This stage is tricky and requires several trials. I find that square rooting `Const_freq` prevents vertices that are very frequent from being too large.

```
> v.size <- sqrt(V(G)$Const_freq)
```

Next, assign a label to each vertex.

```
> v.label <- V(G)$Name
```

Adjust the width of the edges by weighting the edges based on collostruction strength.

```
> E(G)$weight <- E(G)$assoc.chisq
```

Use the `Type` attribute to affect the shape and the color of vertices. Two steps are required: first, convert `V(G)$Type` from character to factor, then from factor to integer.

```
> type <- as.factor(V(G)$Type)
> type <- as.integer(type) # as.numeric() also works
```

You may now select the shapes and colors. Adjectives will be red squares, and verbs dodger blue squares.

```
> v.shapes <- c("circle","square")[type]
> v.colors <- c("red", "dodgerblue")[type]
```

You are almost there. Select the `lgl` layout (the acronym stands for "Large Graph Layout").

```
> l <- layout.lgl(G)
```

In a perfect world, you could plot the graph right now. However, because the graph is large, I recommend that you produce an encapsulated PostScript graphic. This allows you to reduce the size of the graphic file and choose an appropriate scale by selecting a height and a width that are large enough. The `postscript()` function starts the graphics device driver. Its arguments are: the path to the destination file, the orientation of the printed image, whether you allow multiple figures in one file, the size of paper in the printer (`"special"` has to be used when you customize height and width), and `height` and `width`, which set the width and height of the graphics region in inches. The next line is the plot call. As you can see, finding the perfect edge width and arrow size requires several trials! The last line (`dev.off()`) closes the graphics device. Note that the positions of vertices and egdes in your graph will differ from what you have in Fig. B.14. What will not change is the number, size, color, and shape of vertices, and the number, length, orientation, and arrow size of edges.

```
> postscript("/CLSR/chap10/network_ldl.eps", horizontal = FALSE, onefile = FALSE,
+            paper = "special", height=70, width=70)
> plot(G, layout=l, edge.width=(log10(E(G)$weight))^2, edge.arrow.size=log10(E(G)$weight),
+      vertex.size=v.size, vertex.label=v.label, vertex.shape=v.shapes, vertex.color= v.colors,
+      vertex.label.cex=v.size/2, vertex.label.color="black", vertex.label.family="Arial")
> dev.off()
```

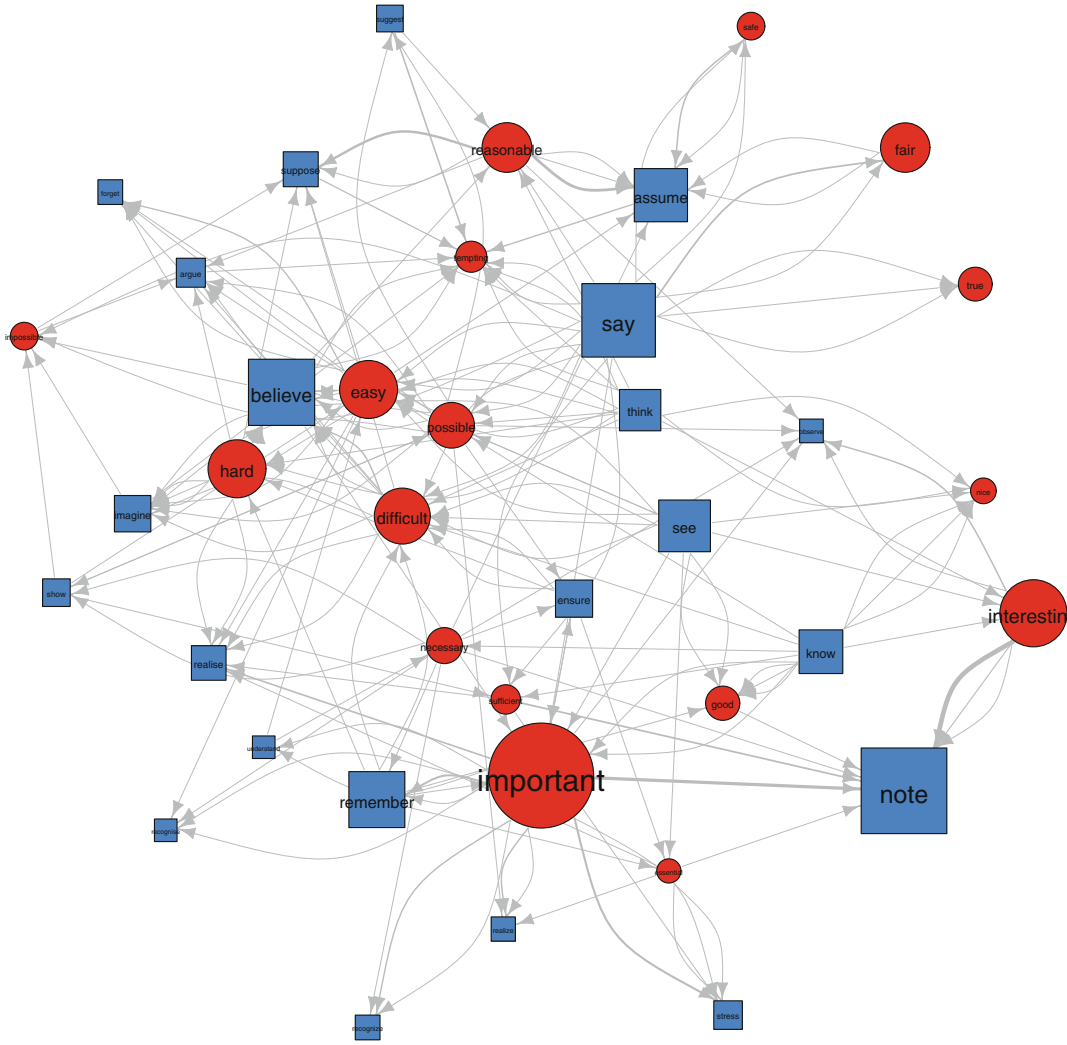


Fig. B.14: A network graph of the constituents of *it BE ADJ to V_inf that* in the BNC

Index

A
approach
  corpus-based, 8–10, 51, 273
  corpus-driven, 9, 10
association measures, 10, 197, 198, 203–205, 207, 209, 212–214, 222, 226, 292
  ΔP, 223–225, 287, 292
  Fisher, 205, 206, 209
  χ^2, 209, 212
  χ^2, 205, 208–213, 291, 292
  log-likelihood ratio, 48, 205, 209
  mutual information, 205, 206, 213

B
backreferencing, 80–82, 98, 106

C
central tendency
  mean, 21, 140–142, 147–149, 169, 176
  median, 140–143, 146–148, 169
  mode, 143, 144
character classes, 70, 77–79
character encoding
  ASCII, 55, 57
  unicode, 55
  UTF, 55–57, 60
character string, 23, 26–28, 32, 43, 44, 54, 61, 69–72, 74–77, 80–83, 85, 89, 90, 97, 101, 102, 109, 114
  substring, 80, 101
cluster analysis, 239
collostructional analysis
  co-varying collexeme analysis (CCA), 213, 214, 219–221
  collexeme analysis, 202, 213–216
  collostructional analysis, 202, 213, 214
  distinctive collexeme analysis (DCA), 202, 213, 214, 216–220
concordance, 7, 10, 87–89, 94, 95, 113, 198
confidence ellipsis, 275–277
contingency table, 153, 154, 179, 194, 203, 204, 206–214, 216, 217, 219, 222–225, 257, 259–261, 278, 291
cooccurrence
  colligation, 198, 200
  collocation, 10, 198–200, 202, 204, 216, 222, 283
  collostruction, 198
  cooccurrence, 198, 278
corpora
  BNC, 4–6, 34–36, 39, 48, 51–54, 58, 59, 65, 70, 82, 87, 88, 95, 96, 102, 104, 106, 107, 110, 112, 113, 116–119, 122, 133, 140, 152, 153, 158, 160, 163, 164, 188–190, 193, 194, 219, 225–227, 253, 254, 270, 275, 290, 291
  BoE, 3, 51, 52
  Brown, 4, 51–53, 186
  CHILDES, 5, 52, 53, 58
  COCA, 6, 34, 39, 51–53, 205, 206, 210, 211, 213, 223, 278
  COHA, 52, 53, 125–129, 134
  COLT, 51–53
  enTenTen12, 6, 52, 53
  FLOB, 52, 53
  Frown, 52, 53
  GloWbE, 34, 35, 52, 53
  Hansard, 39
  Helsinki, 52, 53
  ICE-GB, 52, 53, 217
  Lampeter, 52, 53
  LLC, 52
  LOB, 4, 52, 53, 186
  OANC, 86
  SBCAE, 52
  Strathy, 39, 178
corpus
  annotated, 5, 51, 54, 58, 63, 65, 79, 81, 86, 95, 97, 113, 122, 159, 239, 243, 252, 253
  annotation, 58, 59, 80, 97, 252, 253
  balance, 2–6, 57, 128
  comparability, 3, 181

G. Desagulier, *Corpus Linguistics and Statistics with R*, Quantitative Methods in the Humanities and Social Sciences, DOI 10.1007/978-3-319-64572-8

markup, 6, 54, 58
SGML, 54, 58
XML, 54, 58, 59, 70, 79–81, 95, 97, 188, 193, 194, 253, 270
naturalness, 3, 5, 109, 117
representativeness, 2–6, 51, 57
sampling, 3–5, 7, 71, 95, 109, 151, 157–159, 163, 164, 176, 181, 198, 204, 209, 213, 243, 244, 248, 250
sampling scheme, 3, 4, 57
unannotated, 57, 95
correlation, 186–188, 192, 193
coefficient, 186, 191, 192, 244, 248
Kendall's τ, 186, 189, 191, 192, 194
matrix, 188, 191, 192, 244
pairwise, 188, 189, 245, 246
Pearson's r, 186, 188, 191
Spearman's ρ, 186, 192
spurious, 193
correlation matrix, 194
correspondence analysis (CA), 239, 240, 242, 243, 257, 259–261, 263, 264, 266, 268, 269, 291

D

data frame, 15, 24, 36–42, 48, 101, 104, 107, 113, 116, 120, 125, 130–134, 140, 143, 145, 147, 152, 153, 159, 160, 179, 199, 201, 211, 212, 214, 215, 217, 218, 220, 221, 229, 239, 244, 252–255, 259, 267, 268, 270–272, 276, 281, 291
dispersion
interquartile range (IQR), 145, 146
quantiles, 145–147, 168, 171
standard deviation, 145, 147–149, 169, 245
variance, 147, 148, 176, 243, 245, 247, 248, 263, 264, 272, 279
dissimilarity matrix, 278
distance matrix, 279
distribution
continuous, 164, 169, 177, 209
discrete, 164, 165
probability, 154, 164, 165

E

eigenvalue, 241, 247, 264, 271, 273
empirical cycle, 8, 9

F

`for` loops, 41–44, 82, 90, 91, 95–98, 101–103, 106, 107, 111, 124, 133, 148, 173, 174, 211, 212, 235
factor, 24, 38, 40, 130, 132, 140, 255, 271
frequency, 108, 198, 227, 227, 228, 291
counts, 34, 153, 160, 161, 179
distribution, 48, 145–148, 154, 163, 170, 188, 203
list, 7, 14, 71, 73, 79, 108–110, 112, 114–118, 120–122, 152, 163, 164, 186, 199, 202, 228
raw, 122, 133, 134, 178, 179, 183, 200, 202, 215, 218, 259
relative, 263

H

hierarchical cluster analysis (HCA), 239, 240, 276–278, 280, 291
hypothesis
alternative, 160–163, 165, 168, 169, 171, 172, 176, 179, 188, 191, 192, 208, 209
null, 160–163, 165, 168, 169, 171, 172, 175, 176, 179, 188, 191, 206, 208–210, 212, 213
research, 7, 160
testing, 8, 162, 178, 191

I

`if...else` statements, 43, 44, 92, 93
`if` statements, 43, 44, 92
independence
test of
Fisher exact, 185
χ^2, 178, 179, 183, 185
inertia, 247, 250, 263–265, 271

K

k-means, 255, 257

L

line anchors, 74, 75, 90
list, 15, 24, 33–35, 39, 71, 99–101, 126
LNRE models, 229, 231
lookaround
lookahead
negative, 83
positive, 83
lookbehind
negative, 84
positive, 83–85

M

matching, 31, 70–77, 79–85, 89–91, 95–102, 104–106, 111, 113, 123–125, 129, 130, 159
exact, 81, 85, 86
greedy, 79
lazy, 80, 98, 123
matrix, 15, 24, 34–36, 38, 41, 42, 48, 101, 179, 182, 183, 188, 203, 206, 208, 209, 212, 254, 257, 279
multifactorial analysis, 240, 241, 257
multiple correspondence analysis (MCA), 52, 53, 239, 240, 242, 243, 268–277, 291

N

n-grams, 114, 223
networks
graph theory, 239, 283
network, 10, 239, 253, 283, 289, 292

O
operator
assignment, 20, 21
logical, 31, 32, 110

P
plot
barplot, 117, 133, 144, 179, 272
boxplot, 146, 147, 149
density curve, 169, 170, 172–174
dispersion plot, 122–126, 134, 146
histogram, 117, 119, 170, 173, 194
motion chart, 127, 132–134
strip chart, 125–127, 134, 146
word cloud, 118–122
population, 4, 151, 157, 158, 163, 171, 178
principal component analysis (PCA), 239, 240, 242–245, 248, 250, 252, 254, 266, 291
probability
conditional, 155, 193, 194
joint, 154, 155
marginal, 154
simple, 193

Q
quantifiers, 75, 76, 79

R
regular expressions, 69–71, 73–77, 80–84, 86, 90, 95, 97, 99, 109–111, 130

S
sample size, 147, 182, 186, 229, 231, 261, 263
split, 71, 89, 108, 109, 123
stoplist, 110, 114

T
t-SNE, 254, 256–258
tagging
POS, 58–62, 65, 79–81, 86, 104, 106, 110
semantic, 58
type-token ratio (TTR), 228, 230

V
variable
categorical/nominal, 52, 139, 140, 178, 179, 201, 243, 250, 268, 270, 275, 276
continuous, 139, 164, 250
discrete, 139, 164, 277
explanatory/descriptive/independent, 160
random, 158, 164, 169, 186
response/dependent, 159, 160
vector
character, 24–26, 28–30, 32, 37, 55–57, 70–72, 79, 81, 88, 89, 95, 96, 98, 99, 106, 121, 123, 211
length, 24, 27–33, 37, 42–44, 47, 75, 85, 89, 92
logical, 24–26, 28
numeric, 21, 24–27, 29, 101, 123, 124, 132, 140, 141, 289
vector, 15, 24–35, 39, 40, 42–44, 47, 48, 71–86, 88–92, 97–99, 101, 102, 105, 106, 108, 117, 123–126, 140–142, 147, 148, 176, 245, 286
vocabulary growth curve, 227–234