À mes parents,
À Madeleine

Preface

Preface to the third edition

Preface to the second edition

Eight years, from 2006 to 2014, is a very long time in computer science. The trends I described in the preface of the first edition have not only been confirmed, but accelerated. I tried to reflect this with a complete revision of the techniques exposed in this book: I redesigned or updated all the chapters, I introduced two new ones, and, most notably, I considerably expanded the sections using machine-learning techniques. To make place for them, I removed a few algorithms of lesser interest. This enabled me to keep the size of the book to ca. 700 pages. The programs and companion slides are available from the book web site at `http://ilppp.cs.lth.se/`.

This book corresponds to a course in natural language processing offered at Lund University. I am grateful to all the students who took it and helped me write this new edition through their comments and questions. Curious readers can visit the course site at `http://cs.lth.se/EDAN20/` and see how we use this book in a teaching context.

I would like to thank the many readers of the first edition who gave me feedback or reported errors, the anonymous copy editor of the first and second editions, Richard Johansson and Michael Covington for their suggestions, as well as Peter Exner, the PhD candidate I supervised during this period, for his enthusiasm. Special thanks go to Ronan Nugent, my editor at Springer, for his thorough review and copyediting along with his advice on style and content.

This preface would not be complete without a word to those who passed away, my aunt, Madeleine, and my father, Pierre. There is never a day I do not think of you.

Lund, Pierre Nugues
April 2014

Preface to the first edition

In the past 20 years, natural language processing and computational linguistics have considerably matured. The move has mainly been driven by the massive increase of textual and spoken data and the need to process them automatically. This dramatic growth of available data spurred the design of new concepts and methods, or their improvement, so that they could scale up from a few laboratory prototypes to proven applications used by billions of people. Concurrently, the speed and capacity of machines became an order of magnitude larger, enabling us to process gigabytes of data and billions of words in a reasonable time, to train, test, retrain, and retest algorithms like never before. Although systems entirely dedicated to language processing remain scarce, there are now scores of applications that, to some extent, embed language processing techniques.

The industry trend, as well as the user's wishes, toward information systems able to process textual data has made language processing a new requirement for many computer science students. This has shifted the focus of textbooks from readers being mostly researchers or graduate students to a larger public, from readings by specialists to pragmatism and applied programming. Natural language processing techniques are not completely stable, however. They consist of a mix that ranges from well-mastered and routine to rapidly changing. This makes the existence of a new book an opportunity as well as a challenge.

This book tries to take on this challenge and find the right balance. It adopts a hands-on approach. It is a basic observation that many students have difficulties going from an algorithm exposed using pseudocode to a runnable program. I did my best to bridge the gap and provide the students with programs and ready-made solutions. The book contains real code the reader can study, run, modify, and run again. I chose to write examples in two languages to make the algorithms easy to understand and encode: Perl and Prolog.

One of the major driving forces behind the recent improvements in natural language processing is the increase of text resources and annotated data. The huge amount of texts made available by the Internet and never-ending digitization led many practitioners to evolve from theory-oriented, armchair linguists to frantic empiricists. This books attempts as well as it can to pay attention to this trend and stresses the importance of corpora, annotation, and annotated corpora. It also tries to go beyond English only and expose examples in two other languages, namely French and German.

The book was designed and written for a quarter or semester course. At Lund, I used it when it was still in the form of lecture notes in the EDA171 course. It comes with a companion web site where slides, programs, corrections, an additional chapter, and Internet pointers are available: `http://www.cs.lth.se/~pierre/ilppp/`. All the computer programs should run with Perl (available from www.perl.com) or Prolog. Although I only tested the programs with SWI Prolog available from www.swi-prolog.org, any Prolog compatible with the ISO reference should apply.

Many people helped me during the last 10 years when this book took shape, step-by-step. I am deeply indebted to my colleagues and to my students in classes at Caen,

Nottingham, Stafford, Constance, and now in Lund. Without them, it could never have existed. I would like most specifically to thank the PhD students I supervised, in chronological order, Pierre-Olivier El Guedj, Christophe Godéreaux, Dominique Dutoit, and Richard Johansson.

Finally, my acknowledgments would not be complete without the names of the people I most cherish and who give meaning to my life: my wife, Charlotte, and my children, Andreas and Louise.

Lund, Pierre Nugues
January 2006

Contents

1

An Overview of Language Processing

Γνῶθι σεαυτόν
'Know thyself'
Inscription at the entrance to Apollo's Temple at Delphi

1.1 Linguistics and Language Processing

Linguistics is the study and the description of human languages. Linguistic theories on grammar and meaning have developed since ancient times and the Middle Ages. However, modern linguistics originated at the end of the nineteenth century and the beginning of the twentieth century. Its founder and most prominent figure was probably Ferdinand de Saussure (1916). Over time, modern linguistics has produced an impressive set of descriptions and theories.

Computational linguistics is a subset of both linguistics and computer science. Its goal is to design mathematical models of language structures enabling the automation of language processing by a computer. From a linguist's viewpoint, we can consider computational linguistics as the formalization of linguistic theories and models or their implementation in a machine. We can also view it as a means to develop new linguistic theories with the aid of a computer.

From an applied and industrial viewpoint, language and speech processing, which is sometimes referred to as natural language processing (NLP), natural language understanding (NLU), or language technology, is the mechanization of human language faculties. People use language every day in conversations by listening and talking, or by reading and writing. It is probably our preferred mode of communication and interaction. Ideally, automated language processing would enable a computer to understand texts or speech and to interact accordingly with human beings.

Understanding or translating texts automatically and talking to an artificial conversational assistant are major challenges for the computer industry. Although this final goal has not been reached yet, in spite of constant research, it is being approached every day, step-by-step. Even if we have missed Stanley Kubrick's prediction of talking electronic creatures in the year 2001, language processing and under-

standing techniques have already achieved results ranging from very promising to near-perfect. The description of these techniques is the subject of this book.

1.2 Applications of Language Processing

At first, language processing is probably easier understood by the description of a result to be attained rather than by the analytical definition of techniques. Ideally, language processing would enable a computer to analyze huge amounts of text and to understand them; to communicate with us in a written or a spoken way; to capture our words whatever the entry mode: through a keyboard or through a speech recognition device; to parse our sentences; to understand our utterances, to answer our questions, and possibly to have a discussion with us – the human beings.

Language processing has a history nearly as old as that of computers, and it comprises a large body of work. However, many early attempts remained in the stage of laboratory demonstrations or simply failed. Significant applications have been slow to come, and they are still relatively scarce compared with the universal deployment of some other technologies such as operating systems, databases, and networks. Nevertheless, the number of commercial applications or significant laboratory prototypes embedding language processing techniques is increasing. Examples include:

Spelling and grammar checkers. These programs are now ubiquitous in text processors, and hundred of millions of people use them every day. Spelling checkers are based primarily on computerized dictionaries, and they remove most misspellings that occur in documents. Grammar checkers, although not perfect, have improved to a point that many users could not write a single e-mail without them. Grammar checkers use rules to detect common grammar and style errors (Jensen et al., 1993).

Text indexing and information retrieval from the Internet. These programs are among the most popular of the Web. They are based on crawlers that visit internet sites and that download texts they contain. Crawlers track the links occurring on the pages and thus explore the Web. Many of these systems carry out a full text indexing of the pages. Users ask questions and text retrieval systems return the internet addresses of documents containing words of the question. Using statistics on words or popularity measures, text retrieval systems are able to rank the documents (Salton, 1988; Brin and Page, 1998).

Speech transcription. These systems are based on speech recognition. Instead of typing using a keyboard, speech dictation systems allow a user to dictate reports and transcribe them automatically into a written text. Systems like Microsoft's *Windows Speech Recognition* or Google's *Voice Search* have high performance and recognize English, French, German, Spanish, Italian, Japanese, Chinese, etc. Some systems transcribe radio and TV broadcast news with a word-error rate lower than 10% (Nguyen et al., 2004).

Voice control of domestic devices such as videocassette recorders or disc changers (Ball et al., 1997). These systems are embedded in objects to provide them with

a friendlier interface. Many people find electronic devices complicated and are unable to use them satisfactorily. A spoken interface would certainly be an easier means to control them. Although there are commercial systems available, few of them are fully usable. One challenge they still have to overcome is to operate in noisy environments that impair speech recognition.

Interactive voice response applications. These systems deliver information over the telephone using speech synthesis or prerecorded messages. In more traditional systems, users interact with the application using touch-tone telephones. More advanced servers have a speech recognition module that enables them to understand spoken questions or commands from users. Early examples of speech servers include travel information and reservation services (Mast et al., 1994; Sorin et al., 1995). Although most servers are just interfaces to existing databases and have limited reasoning capabilities, they have spurred significant research on dialogue, speech recognition, and synthesis.

Machine translation. Research on machine translation is one of the oldest domains of language processing. One of its outcomes is the venerable SYSTRAN program that started with translations between English and Russian for the US Department of Defense. Since then, machine translation has been extended to many other languages and has become a mainstream NLP application: *Google Translate* now supports more than 60 languages and is used by more than 200 million people every month (Och, 2012). Another pioneer example is the *Spoken Language Translator* that translated spoken English into spoken Swedish in a restricted domain in real time (Agnäs et al., 1994; Rayner et al., 2000).

Conversational agents. Conversational agents are elaborate dialogue systems that have understanding faculties. An example is TRAINS that helps a user plan a route and the assembling trains: boxcars and engines to ship oranges from a warehouse to an orange juice factory (Allen et al., 1995). Ulysse is another example that uses speech to navigate into virtual worlds (Godéreaux et al., 1996, 1998).

Question answering. Question answering systems reached a milestone in 2011 when *IBM Watson* outperformed all its human contestants in the *Jeopardy!* quiz show (Ferrucci, 2012). Watson answers questions in any domain posed in natural language using knowledge extracted from Wikipedia and other textual sources, encyclopedias, dictionaries, as well as databases such as WordNet, DBpedia, and Yago (Fan et al., 2012).

Some of these applications are widespread, like spelling and grammar checkers. Others are not yet ready for industrial exploitation or are still too expensive for popular use. They generally have a much lower distribution. Unlike other computer programs, results of language processing techniques rarely hit a 100% success rate. Speech recognition systems are a typical example. Their accuracy is assessed in statistical terms. Language processing techniques become mature and usable when they operate above a certain precision and at an acceptable cost. However, common to these techniques is that they are continuously improving and they are rapidly changing our way of interacting with machines.

1.3 The Different Domains of Language Processing

Historically linguistics has been divided into disciplines or levels, which go from sounds to meaning. Computational processing of each level involves different techniques such as signal and speech processing, statistics and machine learning, automaton theory, parsing, first-order logic, and automated reasoning.

A first discipline of linguistics is **phonetics**. It concerns the production and perception of acoustic sounds that form the speech signal. In each language, sounds can be classified into a finite set of **phonemes**. Traditionally, they include **vowels**: *a*, *e*, *i*, *o*; and **consonants**: *p*, *f*, *r*, *m*. Phonemes are assembled into **syllables**: *pa*, *pi*, *po*, to build up the words.

A second level concerns the **words**. The word set of a language is called a **lexicon**. Words can appear in several forms, for instance, the singular and the plural forms. **Morphology** is the study of the structure and the forms of a word. Usually a lexicon consists of root words. Morphological rules can modify or transform the root words to produce the whole vocabulary.

Syntax is a third discipline in which the order of words in a sentence and their relationships is studied. Syntax defines word categories and functions. Subject, verb, object is a sequence of functions that corresponds to a common order in many European languages including English and French. However, this order may vary, and the verb is often located at the end of the sentence in German. **Parsing** determines the structure of a sentence and assigns functions to words or groups of words.

Semantics is a fourth domain of linguistics. It considers the meaning of words and sentences. The concept of "meaning" or "signification" can be controversial. Semantics is differently understood by researchers and is sometimes difficult to describe and process. In a general context, semantics could be envisioned as a medium of our thought. In applications, semantics often corresponds to the determination of the sense of a word or the representation of a sentence in a logical format.

Pragmatics is a fifth discipline. While semantics is related to universal definitions and understandings, pragmatics restricts it – or complements it – by adding a contextual interpretation. Pragmatics is the meaning of words and sentences in specific situations.

The production of language consists of a stream of sentences that are linked together to form a **discourse**. This discourse is usually aimed at other people who can answer – it is to be hoped – through a **dialogue**. A dialogue is a set of linguistic interactions that enables the exchange of information and sometimes eliminates misunderstandings or ambiguities.

1.4 Phonetics

Sounds are produced through vibrations of the vocal cords. Several cavities and organs modify vibrations: the vocal tract, the nose, the mouth, the tongue, and the teeth. Sounds can be captured using a microphone. They result in signals such as that in Fig. 1.1.

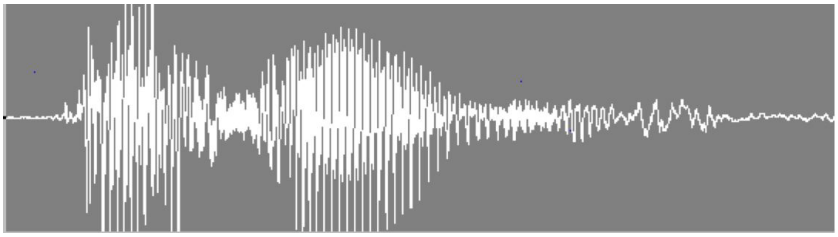

Fig. 1.1. A speech signal corresponding to *This is* [ðɪs ɪz]

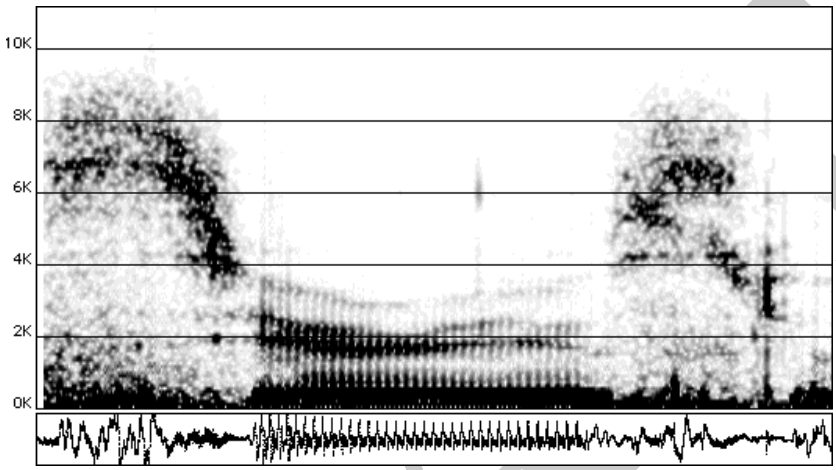

Fig. 1.2. A spectrogram corresponding to the word *serious* [sɪəriəs]

A speech signal can be sampled and digitized by an analog-to-digital converter. It can then be processed and transformed by a Fourier analysis (FFT) in a moving window, resulting in spectrograms (Figs. 1.2 and 1.3). Spectrograms represent the distribution of speech power within a frequency domain ranging from 0 to 10,000 Hz over time. This frequency domain corresponds roughly to the sound production possibilities of human beings.

Phoneticians can "read" spectrograms, that is, split them into a sequence of relatively regular – stationary – patterns. They can then annotate the corresponding segments with phonemes by recognizing their typical patterns.

A descriptive classification of phonemes includes:

- Simple vowels such as /ɪ/, /a/, and /ɛ/, and nasal vowels in French such as /ɑ̃/ and /ɔ̃/, which appear on the spectrogram as a horizontal bar – the fundamental frequency – and several superimposed horizontal bars – the harmonics.
- Plosives such as /p/ and /b/ correspond to a stop in the airflow and then a very short and brisk emission of air from the mouth. The air release appears as a vertical bar from 0 to 5,000 Hz.

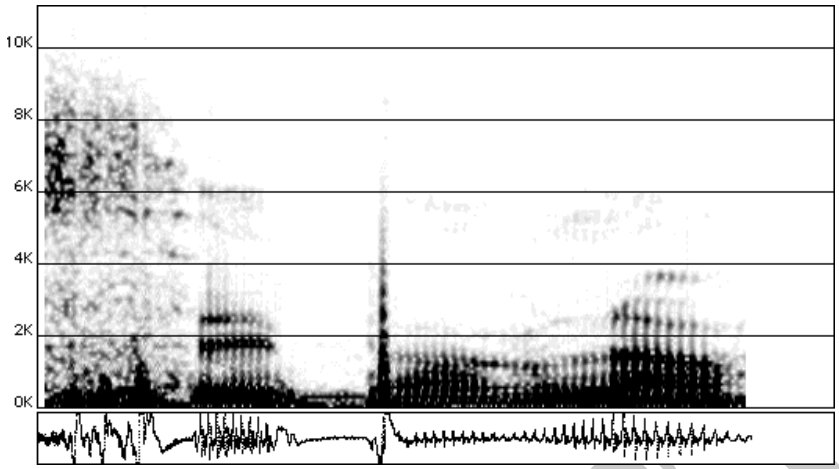

Fig. 1.3. A spectrogram of the French phrase *C'est par là* [separla] 'It is that way'

- Fricatives such as /s/ and /f/ that appear as white noise on the spectrogram, that is, as a uniform gray distribution. Fricatives sound a bit like a loudspeaker with an unplugged signal cable.
- Nasals and approximants such as /m/, /l/, and /r/ are more difficult to spot and are subject to modifications according to their left and right neighbors.

The pronunciation of a word is basically carried out through its **syllables**, phonetic segments formed of a vowel and one or more consonants. These syllables are more or less stressed or emphasized, and are influenced by neighboring syllables.

The general rhythm of the sentence is the **prosody**. Prosody is quite different from English to French and German and is an open subject of research. It is related to the length and structure of sentences, to questions, and to the meaning of the words.

Speech synthesis uses signal processing techniques, phoneme models, and letter-to-phoneme rules to convert a text into speech and to read it in a loud voice. **Speech recognition** does the reverse and transcribes speech into a computer-readable text. It also uses signal processing and statistical techniques including hidden Markov models (HMM) and language models.

1.5 Lexicon and Morphology

The set of available words in a given context makes up a lexicon. It varies from language to language and within a language according to the context or genre: fiction, news, scientific literature, jargon, slang, or gobbledygook. Every word can be classified through a lexical category or **part of speech** such as article, noun, verb, adjective, adverb, conjunction, preposition, or pronoun. Most of the lexical entities

Table 1.1. Grammatical features that modify the form of a word

Features	Values	English	French	German
Number	Singular	*a car*	*une voiture*	*ein Auto*
	Plural	*two cars*	*deux voitures*	*zwei Autos*
Gender	Masculine	*he*	*il*	*er*
	Feminine	*she*	*elle*	*sie*
	Neuter	*it*		*es*
Conjugation and tense	Infinitive	*to work*	*travailler*	*arbeiten*
	Finite	*he works*	*il travaille*	*er arbeitet*
	Gerund	*working*	*travaillant*	*arbeitend*

come from four categories: noun, verb, adjective, and adverb. Other categories such as articles, pronouns, or conjunctions have a limited and stable number of elements. Words in a sentence can be annotated – tagged – with their part of speech.

For instance, the simple sentences in English, French, and German:

The big cat ate the gray mouse
Le gros chat mange la souris grise
Die große Katze ißt die graue Maus

are annotated as:

The/article *big*/adjective *cat*/noun *ate*/verb *the*/article *gray*/adjective *mouse*/noun
Le/article *gros*/adjectif *chat*/nom *mange*/verbe *la*/article *souris*/nom *grise*/adjectif
Die/Artikel *große*/Adjektiv *Katze*/Substantiv *ißt*/Verb *die*/Artikel *graue*/Adjektiv *Maus*/Substantiv

Morphology is the study of how root words and affixes – the **morphemes** – are composed to form words. Morphology can be divided into **inflection** and **derivation**:

- Inflection is the form variation of a word under certain grammatical conditions. In European languages, these conditions consist notably of the number, gender, conjugation, or tense (Table 1.1).
- Derivation combines affixes to an existing root or stem to form a new word. Derivation is more irregular and complex than inflection. It often results in a change in the part of speech for the derived word (Table 1.2).

Most of the inflectional morphology of words can be described through morphological rules, possibly with a set of exceptions. According to these rules, a morphological parser splits each word as it occurs in a text into morphemes – the root word and the affixes. When affixes have a grammatical content, morphological parsers generally deliver this content instead of the raw affixes (Table 1.3).

Morphological parsing operates on single words and does not consider the surrounding words. Sometimes, the form of a word is ambiguous. For instance, *worked*

Table 1.2. Examples of word derivations

	Words	**Derived words**
English	*real*/adjective	*really*/adverb
French	*courage*/noun	*courageux*/adjective
German	*Der Mut*/noun	*mutig*/adjective

Table 1.3. Decomposition of inflected words into a root and affixes

	Words	**Roots and affixes**	**Lemmas and grammatical interpretations**
English	*worked*	*work* + *ed*	*work* + verb + preterit
French	*travaillé*	*travaill* + *é*	*travailler* + verb + past participle
German	*gearbeitet*	*ge* + *arbeit* + *et*	*arbeiten* + verb + past participle

can be found in *he worked* (*to work* and preterit) or *he has worked* (*to work* and past participle). Another processing stage is necessary to remove the ambiguity and to assign (to annotate) each word with a single part-of-speech tag.

A lexicon may simply be a list of all the **inflected** word forms – a wordlist – as they occur in running texts. However, keeping all the forms, for instance, *work*, *works*, *worked*, generates a useless duplication. For this reason, many lexicons retain only a list of canonical words: the **lemmas**. Lemmas correspond to the entries of most ordinary dictionaries. Lexicons generally contain other features, such as the phonetic transcription, part of speech, morphological type, and definition, to facilitate additional processing. Lexicon building involves collecting most of the words of a language or of a domain. Nonetheless, it is probably impossible to build an exhaustive dictionary since new words are appearing every day.

Morphological rules enable us to generate all the word forms from a lexicon. Morphological parsers do the reverse operation and retrieve the word root and its affixes from its inflected or derived form in a text. Morphological parsers use finite-state automaton techniques. Part-of-speech taggers disambiguate the possible multiple readings of a word. They also use finite-state automata or statistical techniques.

1.6 Syntax

Syntax governs the formation of a sentence from words. Syntax is sometimes combined with morphology under the term morphosyntax. Syntax has been a central point of interest of linguistics since the Middle Ages, but it probably reached an apex in the 1970s, when it captured an overwhelming amount of attention in the linguistics community.

1.6.1 Syntax as Defined by Noam Chomsky

Chomsky (1957) had a determining influence in the study of language, and his views still fashion the way syntactic formalisms are taught and used today. Chomsky's theory postulates that syntax is independent from semantics and can be expressed in terms of logic grammars. These grammars consist of a set of rules that describe the sentence structure of a language. In addition, grammar rules can generate the whole sentence set – possibly infinite – of a definite language.

Generative grammars consist of syntactic rules that fractionate a phrase into subphrases and hence describe a sentence composition in terms of phrase structure. Such rules are called **phrase-structure rules**. An English sentence typically comprises two main phrases: a first one built around a noun called the noun phrase, and a second one around the main verb called the verb phrase. Noun and verb phrases are rewritten into other phrases using other rules and by a set of terminal symbols representing the words.

Formally, a grammar describing a very restricted subset of English, French, or German phrases could be the following rule set:

- A **sentence** consists of a **noun phrase** and a **verb phrase**.
- A **noun phrase** consists of an **article** and a **noun**.
- A **verb phrase** consists of a **verb** and a **noun phrase**.

A very limited lexicon of the English, French, or German words could be made of:

- articles such as *the*, *le*, *la*, *der, den*
- nouns such as *boy*, *garçon*, *Knabe*
- verbs such as *hit*, *frappe*, *trifft*

This grammar generates sentences such as:

The boy hit the ball
Le garçon frappe la balle
Der Knabe trifft den Ball

but also incorrect or implausible sequences such as:

The ball hit the ball
*Le balle frappe la garçon
*Das Ball trifft den Knabe

Linguists use an asterisk (*) to indicate an ill-formed grammatical construction or a nonexistent word. In the French and German sentences, the articles must agree with their nouns in gender, number, and case (for German). The correct sentences are:

La balle frappe le garçon
Der Ball trifft den Knaben

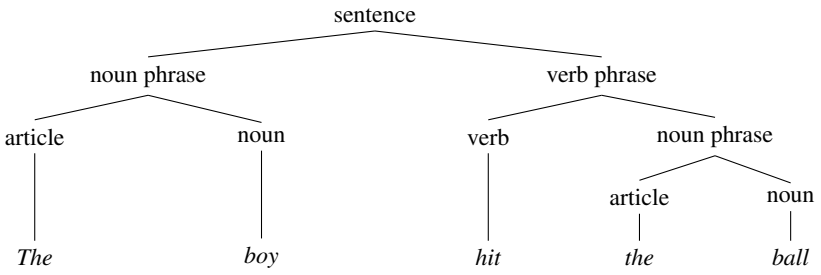

Fig. 1.4. Tree structure of *The boy hit the ball*

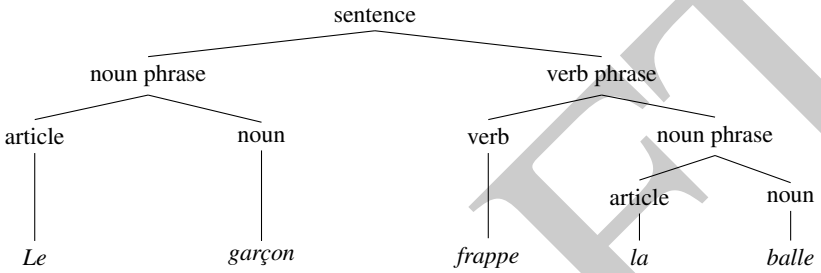

Fig. 1.5. Tree structure of *Le garçon frappe la balle*

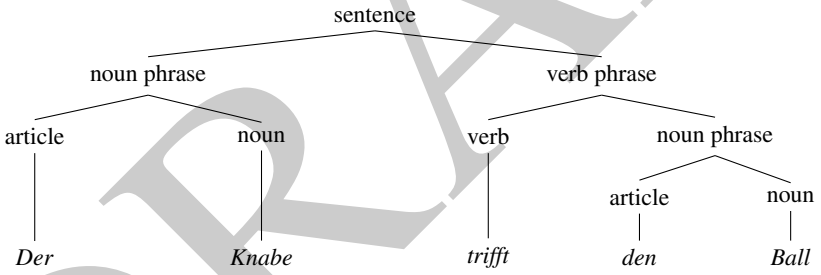

Fig. 1.6. Tree structure of *Der Knabe trifft den Ball*.

Trees can represent the syntactic structure of sentences (Figs. 1.4–1.6) and reflect the rules involved in sentence generation. Moreover, Chomsky's formalism enables some transformations: rules can be set to carry out the building of an interrogative sentence from a declaration, or the building of a passive form from an active one.

Parsing is the reverse of generation, where a grammar, a set of phrase-structure rules, accepts syntactically correct sentences and determines their structure. Parsing requires a mechanism to search the rules that describe the sentence's structure. This mechanism can be applied from the sentence's words up to a rule describing the sentence's structure. This is **bottom-up parsing**. Rules can also be searched from a

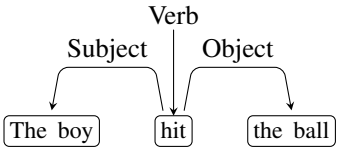

Fig. 1.7. Grammatical relations in the sentence *The boy hit the ball*

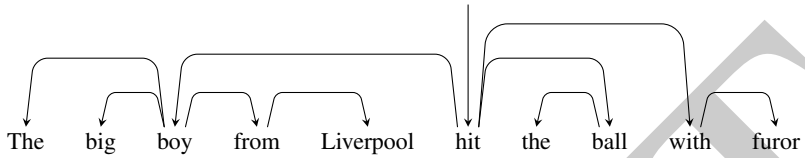

Fig. 1.8. Dependency relations in the sentence *The big boy from Liverpool hit the ball with furor*

sentence structure rule down to the sentence's words. This corresponds to **top-down parsing**.

1.6.2 Syntax as Relations and Dependencies

Before Chomsky, pupils and students learned syntax (and still do so) mainly in terms of functions and relations between the words. A sentence's classical parsing consists in annotating words using parts of speech and in identifying the main verb. The main verb is the pivot of the sentence, and the principal grammatical functions are determined relative to it. Parsing consists then in grouping words to form the subject and the object, which are the two most significant functions in addition to the verb.

In the sentence *The boy hit the ball*, the main verb is *hit*, the subject of *hit* is *the boy*, and its object is *the ball* (Fig. 1.7).

Other grammatical functions (or relations) involve notably articles, adjectives, and adjuncts. We see this in the sentence

The big boy from Liverpool hit the ball with furor.

where the adjective *big* is related to the noun *boy*, and the adjuncts *from Liverpool* and *with furor* are related respectively to *boy* and *hit*.

We can picture these relations as a dependency net, where each word is said to modify exactly another word up to the main verb (Fig. 1.8). The main verb is the head of the sentence and modifies no other word. Tesnière (1966) has extensively described dependency theory.

Recently, **dependency grammars** have enjoyed a growing popularity as they can efficiently handle multiple languages and have a good interface to the semantic level. They provide a theoretical framework to many current parsing techniques and have numerous applications.

Table 1.4. Correspondence between sentences and logical forms

Sentences	Logical forms (predicates)
Pierre wrote notes	`wrote(pierre, notes).`
Pierre a écrit des notes	`a_écrit(pierre, notes).`
Pierre schrieb Notizen	`schrieb(pierre, notizen).`

1.7 Semantics

The semantic level is more difficult to capture, and there are numerous viewpoints on how to define and to process it. A possible viewpoint is to oppose it to syntax: there are sentences that are syntactically correct but that cannot make sense. Such a description of semantics would encompass sentences that make sense. Classical examples by Chomsky (1957) – sentences 1 and 2 – and Tesnière (1966) – sentence 3 – include:

1. *Colorless green ideas sleep furiously.*
2. **Furiously sleep ideas green colorless.*
3. *Le silence vertébral indispose la voile licite.*
   'The vertebral silence embarrasses the licit sail.'

Sentences 1 and 3 and are syntactically correct but have no meaning, while sentence 2 is neither syntactically nor semantically correct.

In computational linguistics, semantics is often related to logic and to predicate calculus. Determining the semantic representation of a sentence then involves turning it into a predicate–argument structure, where the predicate is the main verb and the arguments correspond to phrases accompanying the verb such as the subject and the object. This type of logical representation is called a **logical form**. Table 1.4 shows examples of sentences together with their logical forms.

Representation is only one facet of semantics. Once sentence representations have been built, they can be interpreted to check what they mean. *Notes* in the sentence *Pierre wrote notes* can be linked to a dictionary **definition**. If we look up *notes* in the *Cambridge International Dictionary of English* (Procter, 1995), we find as many as five possible senses for it (abridged from p. 963):

1. **note** [WRITING], *noun*, a short piece of writing;
2. **note** [SOUND], *noun*, a single sound at a particular level;
3. **note** [MONEY], *noun*, a piece of paper money;
4. **note** [NOTICE], *verb*, to take notice of;
5. **note** [IMPORTANCE], *noun*, of note: of importance.

So linking a word meaning to a definition is not straightforward because of possible ambiguities. Among these definitions, the intended sense of *notes* is a specialization of the first entry:

notes, *plural noun*, notes are written information.

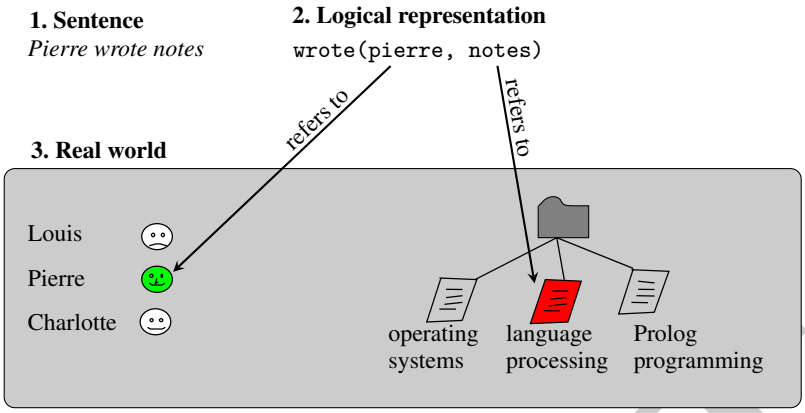

Fig. 1.9. Resolving references of *Pierre wrote notes*

Finally, we can interpret *notes* as what they refer to concretely, that is, a specific object: a set of bound paper sheets with written text on them or a file on a computer drive. The word *notes* is then the mention of an object of the real world, here a file on a computer, and linking the mention and the object is called **reference resolution**.

The **referent** of the word *notes*, that is, the designated object or **entity**, could be the path `/users/pierre/language_processing.html` in Unix parlance. As for the definition of a word, the designated entity can be ambiguous. Let us suppose that a database contains the locations of the lecture notes Pierre wrote. In a predicate–argument structure format, listing its content could yield:

```
notes('/users/pierre/operating_systems.html').
notes('/users/pierre/language_processing.html').
notes('/users/pierre/prolog_programming.html').
```

Here this would mean that finding the referent of *notes* consists in choosing a document among three possible ones (Fig. 1.9).

Obtaining the semantic structure of a sentence has been discussed abundantly in the literature. This is not surprising, given the uncertain nature of semantics. Building a logical form sometimes calls on the **composition** of the semantic representation of the phrases that constitute a sentence. To carry it out, we must assume that sentences and phrases have an internal representation that can be expressed in terms of a logical formula.

Once a representation has been built, a reasoning process is applied to resolve references and to determine whether a sentence is true or not. It generally involves rules of deduction, or **inferences**.

Pragmatics is semantics restricted to a specific context and relies on facts that are external to the sentence. These facts contribute to the inference of a sentence's meaning or prove its truth or falsity. For instance, the pragmatics of

Methuselah lived to be 969 years old. (Genesis 5:27)

can make sense in the Bible but not elsewhere, given the current possibilities of medicine.

1.8 Discourse and Dialogue

An interactive conversational agent cannot be envisioned without considering the whole **discourse** of (human) users – or parts of it – and apart from a **dialogue** between a user and the agent. Discourse refers to a sequence of sentences, to a sentence context in relation with other sentences, or with some background situation. It is often linked with pragmatics.

Discourse study also enables us to resolve references that are not self-explainable in single sentences. Pronouns are good examples of such missing information. In the sentence

John took it

the pronoun *it* can probably be related to an entity mentioned in a previous sentence, or is obvious given the context where this sentence was said. These references are given the name of **anaphors**.

Dialogue provides a means of communication. It is the result of two intermingled – and, we hope, interacting – discourses: one from the user and the other from the machine. It enables a conversation between the two parties, the assertion of new results, and the cooperative search for solutions.

Dialogue is also a tool to repair communication failures or to complete interactively missing data. It may clarify information and mitigate misunderstandings that impair communication. Through a dialogue a computer can respond and ask the user:

I didn't understand what you said! Can you repeat (rephrase)?

Dialogue easily replaces some hazardous guesses. When an agent has to find the potential reference of a pronoun or to solve reference ambiguities, the best option is simply to ask the user to clarify what s/he means:

Tracy? Do you mean James' brother or your mother?

Discourse processing splits texts and sentences into segments. It then sets links between segments to chain them rationally and to map them onto a sort of structure of the text. Discourse studies often make use of **rhetoric** as a background model of this structure.

Dialogue processing classifies the segments into what are called **speech acts**. At a first level, speech acts comprise dialogue turns: the user turn and the system turn. Then turns are split into sentences, and sentences into questions, declarations, requests, answers, etc. Speech acts can be modeled using finite-state automata or more elaborate schemes using **intention** and **planning** theories.

1.9 Why Speech and Language Processing Are Difficult

So far, for all the linguistic levels mentioned in the previous sections, we outlined models and techniques to process speech and language. They often enable machines to obtain excellent results compared to the performance of human beings. However, for most levels, language processing rarely hits the ideal score of 100%. Among the hurdles that often prevent the machine from reaching this figure, two recur at any level: ambiguity and the absence of a perfect model.

1.9.1 Ambiguity

Ambiguity is a major obstacle in language processing, and it may be the most significant. Although as human beings we are not aware of it most of the time, ambiguity is ubiquitous in language and plagues any stage of automated analysis. We saw examples of ambiguous morphological analysis and part-of-speech annotation, word senses, and references. Ambiguity also occurs in speech recognition, parsing, anaphora solving, and dialogue.

McMahon and Smith (1996) illustrate strikingly ambiguity in speech recognition with the sentence

The boys eat the sandwiches.

Speech recognition comprises generally two stages: first, a phoneme recognition, and then a concatenation of phoneme substrings into words. Using the International Phonetic Association (IPA) symbols, a perfect phonemic transcription of this utterance would yield the transcription:

[ˈðəbˈoɪzˈiːtˈðəsˈændwɪdʒɪz],

which shows eight other alternative readings at the word decoding stage:

*The boy seat the sandwiches.
*The boy seat this and which is.
*The boys eat this and which is.
The buoys eat the sandwiches.
*The buoys eat this and which is.
The boys eat the sand which is.
*The buoys seat this and which is.

This includes the strange sentence

The buoys eat the sand which is.

For syntactic and semantic layers, a broad classification occurs between lexical and structural ambiguity. Lexical ambiguity refers to multiple senses of words, while structural ambiguity describes a parsing alternative, as with the frequently quoted sentence

I saw the boy with a telescope,

which can mean either that I used a telescope to see the boy or that I saw the boy who had a telescope.

A way to resolve ambiguity is to use a conjunction of language processing components and techniques. In the example given by McMahon and Smith, five out of eight possible interpretations are not grammatical. These are flagged with an asterisk. A further syntactic analysis could discard them.

Probabilistic models of word sequences can also address disambiguation. Statistics on word occurrences drawn from large quantities of texts – corpora – can capture grammatical as well as semantic patterns. Improbable alternatives <boys eat sand> and <buoys eat sand> are also highly unlikely in corpora and will not be retained (McMahon and Smith, 1996). In the same vein, probabilistic parsing is a very powerful tool to rank alternative parse trees, that is, to retain the most probable and reject the others.

In some applications, logical rules model the context, reflect common sense, and discard impossible configurations. Knowing the physical context may help disambiguate some structures, as in the boy and the telescope, where both interpretations of the isolated sentence are correct and reasonable. Finally, when a machine interacts with a user, it can ask her/him to clarify an ambiguous utterance or situation.

1.9.2 Models and Their Implementation

Processing a linguistic phenomenon or layer starts with the choice or the development of a formal model and its algorithmic implementation. In any scientific discipline, good models are difficult to design. This is specifically the case with language. Language is closely tied to human thought and understanding, and in some instances models in computational linguistics also involve the study of the human mind. This gives a measure of the complexity of the description and the representation of language.

As noted in the introduction, linguists have produced many theories and models. Unfortunately, few of them have been elaborate enough to encompass and describe language effectively. Some models have also been misleading. This explains somewhat the failures of early attempts in language processing. In addition, many of the potential theories require massive computing power. Processors and storage able to support the implementation of complex models with substantial dictionaries, corpora, and parsers were not widely available until recently.

However, in the last decade models have matured, and computing power has become inexpensive. Although models and implementations are rarely (never?) perfect, they now enable us to obtain exploitable results. Most use a limited set of techniques that we will consider throughout this book, namely finite-state automata, logic grammars, and first-order logic. These tools are easily implemented in Prolog. Another set of tools pertains to the theory of probability, statistics, and machine learning. The combination of logic, statistics and machine-learning techniques now enables us to parse running-text sentences in multiple languages with an accuracy rate of more than 90%, a figure that would have been unimaginable 15 years ago.

Table 1.5. An excerpt of a Persona dialogue. After Ball et al. (1997)

Turns	Utterances
	[Peedy is asleep on his perch]
User:	Good morning, Peedy.
	[Peedy rouses]
Peedy:	**Good morning.**
User:	Let's do a demo.
	[Peedy stands up, smiles]
Peedy:	**Your wish is my command, what would you like to hear?**
User:	What have you got by Bonnie Raitt?
	[Peedy waves in a stream of notes, and grabs one as they rush by.]
Peedy:	**I have "The Bonnie Raitt Collection" from 1990.**
User:	Pick something from that.
Peedy:	**How about "Angel from Montgomery"?**
User:	Sounds good.
	[Peedy drops note on pile]
Peedy:	**OK.**
User:	Play some rock after that.
	[Peedy scans the notes again, selects one]
Peedy:	**How about "Fools in Love"?**
User:	Who wrote that?
	[Peedy cups one wing to his 'ear']
Peedy:	**Huh?**
User:	Who wrote that?
	[Peedy looks up, scrunches his brow]
Peedy:	**Joe Jackson**
User:	Fine.
	[Drops note on pile]
Peedy:	**OK.**

1.10 An Example of Language Technology in Action: the Persona Project

1.10.1 Overview of Persona

The Persona prototype from Microsoft Research (Ball et al., 1997) illustrates a user interface that is based on a variety of language processing techniques. Persona is a conversational agent that helps a user select songs and music tracks from a record database. Peedy, an animated cartoonlike parrot, embodies the agent that interacts with the user. It contains speech recognition, parsing, and semantic analysis modules to listen and to respond to the user and to play the songs. Table 1.5 shows an example of a dialogue with Peedy.

Certain interactive talking assistants consider a limited set of the linguistic levels we have presented before. Simple systems bypass syntax, for example, and have only a speech recognition device to detect a couple of key words. In contrast, Persona

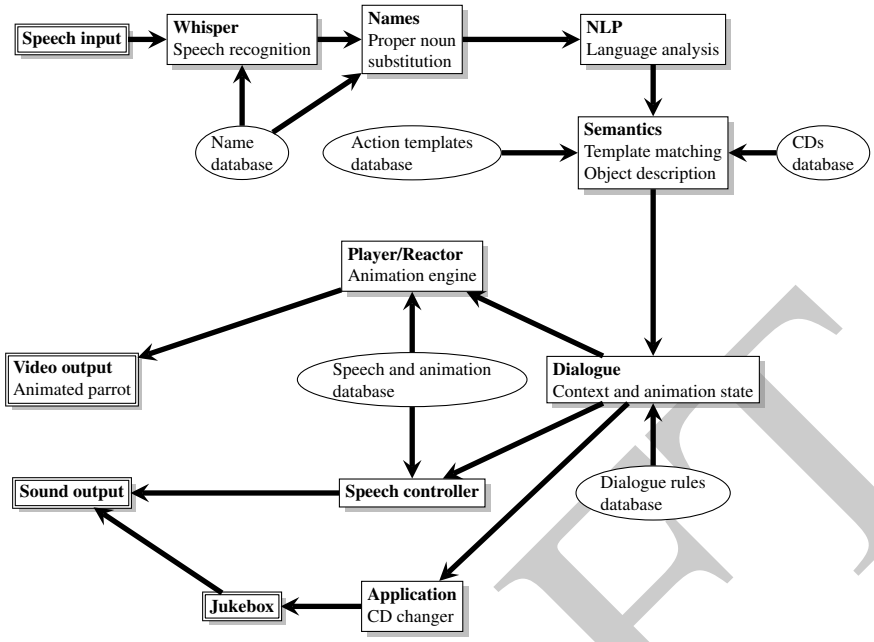

Fig. 1.10. Architecture of the Persona conversational assistant. After Ball et al. (1997)

has components to process more layers. They are organized in modules carrying out speech recognition, speech synthesis, parsing, semantics analysis, and dialogue. In addition, Persona has components specific to the application such as a name substitution module to find proper nouns like *Madonna* or *Debussy* and an animation module to play the Peedy character.

Persona's architecture organizes its modules into a pipeline processing flow (Fig. 1.10). Many other instances of dialogue systems adopt a similar architecture.

1.10.2 The Persona's Modules

Persona's first component is the Whisper speech recognition module (Huang et al., 1995). Whisper uses signal processing techniques to compare phoneme models to the acoustic waves, and it assembles the recognized phonemes into words. It also uses a grammar to constrain the recognition possibilities. Whisper transcribes continuous speech into a stream of words in real time. It is a speaker-independent system. This means that it operates with any speaker without training.

The user's orders to select music often contain names: artists, titles of songs, or titles of albums. The Names module extracts them from the text before they are passed on to further analysis. Names uses a pattern matcher that attempts to substitute all the names and titles contained in the input sentence with placeholders. The utterance *Play before you accuse me by Clapton* is transformed into *Play* `track1` *by* `artist1`.

The NLP module parses the input in which names have been substituted. It uses a grammar with rules similar to that of Sect. 1.6.1 and produces a tree structure. It creates a logical form whose predicate is the verb, and the arguments are the subject and the object: `verb(subject, object)`. The sentence *I would like to hear something* is transformed into the form `like(i, hear(i, something))`.

The logical forms are converted into a task graph representing the utterance in terms of actions the agent can do and objects of the task domain. It uses an application-dependent notation to map English words to symbols. It also reverses the viewpoint from the user to the agent. The logical form of *I would like to hear something* is transformed into the task graph: `verbPlay(you, objectTrack)` – *You play* (`verbPlay`) *a track* (`objectTrack`).

Each possible request Peedy understands has possible variations – paraphrases. The mapping of logical forms to task graphs uses transformation rules to reduce them to a limited set of 17 canonical requests. The transformation rules deal with synonyms, syntactic variation, and colloquialisms. The forms corresponding to

I'd like to hear some Madonna.
I want to hear some Madonna.
It would be nice to hear some Madonna.

are transformed into a form equivalent to

Let me hear some Madonna.

The resulting graph is matched against actions templates the jukebox can carry out.

The dialogue module controls Peedy's answers and reactions. It consists of a state machine that models a sequence of interactions. Depending on the state of the conversation and an input event – what the user says – Peedy will react: trigger an animation, utter a spoken sentence or play music, and move to another conversational state.

1.11 Further Reading

Introductory textbooks on linguistics include *An Introduction to Language* (Fromkin et al., 2010) and *Linguistics: An Introduction to Linguistics Theory* (Fromkin, 2000). The *Nouveau dictionnaire encyclopédique des sciences du langage* (Ducrot and Schaeffer, 1995) is an encyclopedic presentation of linguistics in French, and *Studienbuch Linguistik* (Linke et al., 2004) is an introduction in German. *Fondamenti di linguistica* (Simone, 2007) is an outstandingly clear and concise work in Italian that describes most fundamental concepts of linguistics.

Concepts and theories in linguistics evolved continuously from their origins to the present time. Historical perspectives are useful to understand the development of central issues. *A Short History of Linguistics* (Robins, 1997) is a very readable introduction to linguistics history. *Histoire de la linguistique de Sumer à Saussure* (Malmberg, 1991) and *Analyse du langage au XX^{e} siècle* (Malmberg, 1983) are comprehensive and accessible books that review linguistic theories from the ancient Near

East to the end of the 20th century. *Landmarks in Linguistic Thought, The Western Tradition from Socrates to Saussure* (Harris and Taylor, 1997) are extracts of founding classical texts followed by a commentary.

Available books on natural language processing include (in English): *Natural Language Processing in Prolog* (Gazdar and Mellish, 1989), *Prolog for Natural Language Analysis* (Gal et al., 1991), *Natural Language Processing for Prolog Programmers* (Covington, 1994), *Natural Language Understanding* (Allen, 1994), *Foundations of Statistical Natural Language Processing* (Manning and Schütze, 1999), *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (Jurafsky and Martin, 2008), *Foundations of Computational Linguistics: Human–Computer Communication in Natural Language* (Hausser, 2014). Available books in French include: *Prolog pour l'analyse du langage naturel* (Gal et al., 1989), *L'intelligence artificielle et le langage* (Sabah, 1990). And in German *Grundlagen der Computerlinguistik. Mensch-Maschine-Kommunikation in natürlicher Sprache* (Hausser, 2000).

The Internet offers a wealth of resources: digital libraries, general references, corpus, lexical, and software resources, together with registries and portals. A starting point is the official home page of the Association for Computational Linguistics (ACL), which provides many links (`http://www.aclweb.org/`). The ACL anthology (`http://www.aclweb.org/anthology/`) is an extremely valuable anthology of research papers (journal and conferences) published under the auspices of the ACL. The French journal *Traitement automatique des langues* is also a source of interesting papers. It is published by the Association de traitement automatique des langues (`http://www.atala.org/`). Wikipedia (`http://www.wikipedia.org/`) is a free encyclopedia that contains definitions and general articles on concepts and theories used in computational linguistics and natural language processing.

Many source programs are available for download, either for free or under a license. They include speech synthesis and recognition, morphological analysis, parsing, and so on. The Natural Language Toolkit (NLTK) is an example that features a comprehensive suite of open source Python programs, data sets, and tutorials (`http://nltk.org/`). It has a companion book: *Natural Language Processing with Python* by Bird et al. (2009). The German Institute for Artificial Intelligence Research maintains a list of available software and related resources at the Natural Language Software Registry (`http://registry.dfki.de/`).

Lexical and corpus resources are now available in many languages. Valuable sites include the Linguistic Data Consortium of the University of Pennsylvania (`http://www.ldc.upenn.edu/`) and the European Language Resources Association (`http://www.elra.info/`).

There are nice interactive online demonstrations covering speech synthesis, parsing, translation, and so on. Since sites are sometimes transient, we do not list them here. A good way to find them is to use search engines or directories like Google, Bing, or Yahoo.

Finally, some companies and laboratories are very active in language processing research. They include major software powerhouses like Google, IBM, Microsoft,

Yahoo, and Xerox. The paper describing the Peedy animated character can be found at the Microsoft Research website (`http://www.research.microsoft.com/`).

Exercises

1.1. List some computer applications that are relevant to the domain of language processing.

1.2. Tag the following sentences using parts of speech you know:

The cat caught the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.3. Give the morpheme list of: *sings*, *sung*, *chante*, *chantiez*, *singt*, *sang*. List all the possible ambiguities.

1.4. Give the morpheme list of: *unpleasant*, *déplaisant*, *unangenehm*.

1.5. Draw the tree structures of the sentences:

The cat caught the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.6. Identify the main functions of these sentences and draw the corresponding dependency graph linking the words:

The cat caught the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.7. Draw the dependency graphs of the sentences:

The mean cat caught the gray mouse on the table.
Le chat méchant attrape la souris grise sur la table.
Die böse Katze fängt die graue Maus auf dem Tisch.

1.8. Give examples of sentences that are:

- Syntactically incorrect
- Syntactically correct
- Syntactically and semantically correct

1.9. Give the logical forms of these sentences:

The cat catches the mouse.
Le chat attrape la souris.
Die Katze fängt die Maus.

1.10. List the components you think necessary to build a spoken dialogue system.

2

A Tour of Python

2.1 Why Python?

Python has become the most popular scripting language. Perl, Ruby, or Lua have similar qualities and goals, sport active developer communities, and have application niches. Nonetheless, none of them can claim the spread and universality of Python. Python's rigorous design, ascetic syntax, simplicity, and the availability of scores of libraries made it the language chosen by almost 70% of the American universities for teaching computer science (Guo, 2014). This makes Python unescapable when it comes to natural language processing.

We used Perl in the first editions of this book as it featured rich regular expressions and a support for Unicode; they are still unsurpassed. Python later adopted these features to a point that now makes the lead of Perl in these areas less significant. And the programming style conveyed by Python, both Spartan and elegant, eventually prevailed. The purpose of this chapter is to provide a quick introduction to Python's syntax to readers with some knowledge in programming.

Python comes in two flavors: Python 2 and Python 3. In this book, we only use Python 3 as Python 2 does not properly support Unicode. Moreover, given a problem, there are often many ways to solve it. Among the possible constructs, some are more conformant to the spirit of Python. van Rossum et al. (2013) wrote a guide on the Pythonic coding style that we try to follow in this book.

2.2 The Read, Evaluate, and Print Loop

Once installed, we start Python either from a terminal or an integrated development environment (IDE). Python uses a loop that reads the user's statements, evaluates them, and prints the results (REPL). Python uses a prompt, the `>>>` sequence, to tell it is ready to accept a command. Here is an example, where we create variables and assign them with values, numbers and strings, and carry out a few arithmetic operations:

```
$ python
>>> a = 1 <------------------ We create variable a and assign it with 1
>>> b = 2 <------------------ We create b and assign it with 2
>>> b + 1 <------------------ We add 1 to b
3 <---------------------------- And Python returns the result
>>> c = a / (b + 1) <-------- We carry out a computation and assign it to c
>>> c <---------------------- We print c
0.3333333333333333
>>> text = 'Result:' <------- We create text and assign it with a string
>>> print(text, c) <--------- And we print both text and c
Result: 0.3333333333333333
>>> quit()
$
```

We can also write these statements in a file, `first.py`:

```
# A first program
a = 1
b = 2
c = a / (b + 1)
text = 'Result:'
print(text, c)
```

and execute it by typing:

```
$ python first.py
Result: 0.3333333333333333
```

2.3 Introductory Programs

Like all the structured languages, programs in Python consist of blocks, i.e. bodies of contiguous statements together with control statements. In Python, these blocks are defined by an identical indentation: We create a new block by adding an indentation of four spaces from the previous line. This indentation is decreased by the same number of spaces to mean the end of the block.

The program below uses a loop to print the numbers of a list. The loop starts with the `for` and `in` statements ended with a colon. After this statement, we add an indentation of four spaces to define the body of the loop: The statements executed by this loop. We remove the indentation when the block has ended:

```
for i in [1, 2, 3, 4, 5, 6]:
    print(i)
print('Done')
```

Table 2.1. Summary of the main Python operators

Unary operators	`not`	Logical not
	`~`	Binary not
	`+` and `-`	Arithmetic plus sign and negation
Arithmetic operators	`*`, `/`, `**`	Multiplication, division, and exponentiation
	`//` and `%`	Floor division and modulo
	`+` and `-`	Addition and subtraction
String operator	`+`	String concatenation
Comparison operators	`>` and `<`	Greater than and less than
	`>=` and `<=`	Greater than or equal and less than or equal
	`==` and `!=`	Equal and not equal
Logical operators	`and`	Logical and
	`or`	Logical or
Shift operators	`<<` and `>>`	Shift left and shift right
Binary bitwise operators	`&`, `|`, `^`	and, or, xor

The next program introduces a condition with the `if` and `else` statements, also ended with a colon, and the modulo operator, `%`, to print the odd and even numbers:

```
for i in [1, 2, 3, 4, 5, 6]:
    if i % 2 == 0:
        print('Even:', i)
    else:
        print('Odd:', i)
print('Done')
```

Table 2.1 shows common operators in Python.

2.4 Strings

A string in Python is a sequence of characters or symbols enclosed within matching single, double, or triple quotes as, respectively, `'my string'`, `"my string"`, and `"""my string"""`. We use triple quotes to create strings spanning multiple lines as with:

```
iliad = """Sing, O goddess, the anger of Achilles son of
Peleus, that brought countless ills upon the Achaeans."""
```

where the string is stored in the `iliad` variable.

In the example above, the string includes a new line delimiter, `'\n'`, between *of* and *Peleus* to break the line. If, instead, we want to keep the white spaces and just wrap the line so that it fits our text editor, we will use the backslash continuation character, `\`, as in:

```
iliad2 = 'Sing, O goddess, the anger of Achilles son of \
Peleus, that brought countless ills upon the Achaeans.'
```

where the line break is ignored and `iliad2` is equivalent to one single line. We can use any type of quote then.

2.4.1 String Index

We access the characters in a string using their index enclosed in square brackets, starting at 0:

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
alphabet[0]          # 'a'
alphabet[1]          # 'b'
alphabet[25]         # 'z'
```

We can use negative indices, that start from the end of the string:

```
alphabet[-1]         # the last character of a string: 'z'
alphabet[-2]         # the second last: 'y'
alphabet[-26]        # 'a'
```

An index outside the range of the string, like `alphabet[27]`, will throw an index error.

The length of a string is given by the `len()` function:

```
len(alphabet)        # 26
```

There is no limit to this length; we can use them to store a whole corpus, provided that our machine has enough memory.

Once created, strings are immutable and we cannot change their content:

```
alphabet[0] = 'b'    # throws an error
```

2.4.2 String Operations and Functions

Strings come with a set of built-in operators and functions. We concatenate and repeat strings using `+` and `*` as in:

```
'abc' + 'def'    # 'abcdef'
'abc' * 3        # 'abcabcabc'
```

The `join()` function is an alternative to `+`. It is called by a string with a list as argument: `str.join(list)`. It concatenates the elements of the list with the calling string, possibly empty, placed in-between:

```
''.join(['abc', 'def', 'ghi'])   # equivalent to a +:
                                 # 'abcdefghi'
' '.join(['abc', 'def', 'ghi'])  # places a space between the
                                 # elements: 'abc def ghi'
', '.join(['abc', 'def', 'ghi']) # 'abc, def, ghi'
```

We set a string in uppercase letters with `str.upper()` and in lowercase with `str.lower()`:

```
accented_e = 'eéèêë'
accented_e.upper()     # 'EÉÈÊË'
accented_E = 'EÉÈÊË'
accented_E.lower()     # 'eéèêë'
```

We search and replace substrings in strings using `str.find()` and `str.replace()`. `str.find()` returns the index of the first occurrence of the substring or -1, if not found, while `replace()` replaces all the occurrences of the substring and returns a new string:

```
alphabet.find('def')        # 3
alphabet.find('é')          # -1
alphabet.replace('abc', 'αβγ') # 'αβγdefghijklmnopqrstuvwxyz'
```

We can iterate over the characters of a string using a `for in` loop, and for instance extract all its vowels as in:

```
text_vowels = ''
for i in iliad:
    if i in 'aeiou':
        text_vowels = text_vowels + i
print(text_vowels)   # 'ioeeaeoieooeeuaououeiuoeaea'
```

We can abridge the statement:

```
text_vowels = text_vowels + i
```

into

```
text_vowels +=  i
```

as well as for all the arithmetic operators: `-=`, `*=`, `/=`, `**=`, and `%=`.

2.4.3 Slices

We can extract substrings of a string using **slices**: A range defined by a start and an end index, `[start:end]`, where the slice will include all the characters from index `start` up to index `end - 1`:

```
alphabet[0:3]    # the three first letters of alphabet: 'abc'
alphabet[:3]     # equivalent to alphabet[0:3]
alphabet[3:6]    # substring from index 3 to index 5: 'def'
alphabet[-3:]    # the three last letters of alphabet: 'xyz'
alphabet[10:-10] # 'klmnop'
alphabet[:]      # all the letters: 'a...z'
```

As the end index is excluded from the slice,

```
alphabet[:i] + alphabet[i:]
```

Table 2.2. Escape sequences in Python

Sequence	Description	Sequence	Description
`\t`	Tabulation	`\100`	Octal ASCII, three digits, here @
`\n`	New line	`\x40`	Hexadecimal ASCII, two digits, here @
`\r`	Carriage return	`\N{COMMERCIAL AT}`	Unicode name, here @
`\f`	Form feed	`\u0152`	Unicode code point, 16 bits, here Œ
`\b`	Backspace	`\U00000152`	Unicode code point, 32 bits, here Œ
`\a`	Bell		
`\'`	Single quote		
`\"`	Double quote		
`\\`	Backslash		

is always equal to the original string, whatever the value of `i`.

In addition to the start and the end, we can add a step using the syntax `[start:end:step]`. With a step of 2, we extract every second letter:

```
alphabet[0::2]   # acegikmoqzuwy
```

2.4.4 Special Characters

The characters in the strings are interpreted literally by Python, except the quotes and backslashes. To create strings containing these two characters, Python defines two *escape sequences*: `\'` to represent a quote and `\\` to represent a backslash as in:

```
'Python\'s strings'    # "Python's strings"
```

This expression creates the string *Python's strings*; the backslash escape character tells Python to read the quote literally instead of interpreting it as an end-of-string delimiter.

We can also use literal single quotes inside a string delimited by double quotes as in:

```
"Python's strings"     # "Python's strings"
```

Python interpolates certain backslashed sequences, like `\n` or `\t`. For example, `\n` is interpreted as a new line and `\t` as a tabulation. Table 2.2 shows a list of escape sequences with their meaning.

The right column in Table 2.2 lists the numerical representations of characters using the ASCII and Unicode standards. The `\N{name}` name and `\uxxxx` and `\Uxxxxxxxx` sequences enable us to designate any character, like *Ö* and *Œ*, by its Unicode name, respectively, `\N{LATIN CAPITAL LETTER O WITH DIAERESIS}` and `\N{LATIN CAPITAL LIGATURE OE}`, or its code point, `\u00D6` and `\u0152`. We review both the ASCII and Unicode schemes in Chap. 4. We can also use `\ooo` octal and `\xhh` hexadecimal sequences:

```
'\N{COMMERCIAL AT}'    # '@'
'\x40'                 # '@'
'\100'                 # '@'
'\u0152'               # 'Œ'
```

If we want to treat backslashes as normal characters, we add the `r` prefix (raw) to the string as in:

```
r'\N{COMMERCIAL AT}'    # '\\N{COMMERCIAL AT}'
r'\x40'                 # '\\x40'
r'\100'                 # '\\100'
r'\u0152'               # '\\u0152'
```

These raw strings will be useful to write regular expressions; see Sect. 3.3.

2.4.5 Formatting Strings

Python can interpolate variables inside strings. This process is called formatting and uses the `str.format()` function. The positions of the variables in the string are given by curly braces: `{}` that will be replaced by the arguments in `format()` in the same order as in:

```
begin = 'my'
'{} string {}'.format(begin, 'is empty')
                                    # 'my string is empty'
```

`format()` has many options like reordering the arguments through indices:

```
begin = 'my'
'{1} string {0}'.format('is empty', begin)
                                    # 'my string is empty'
```

If the input string contains braces, we escape them by doubling them: `{{` for a literal `{` and `}}` for `}`.

2.5 Data Types

Python has a rich set of data types. The primitive types include:

- The Boolean type, `bool`, with the values `True` and `False`;
- The None type with the `None` value as unique member, equivalent to null in C or Java;
- The integers, `int`;
- The floating point numbers, `float`.

We have also seen the `str` string data type consisting of sequences of Unicode characters.

We return the type of a value with the `type()` function:

```
type(alphabet)     # <class 'str'>
type(12)           # <class 'int'>
type('12')         # <class 'str'>
type(12.0)         # <class 'float'>
type(True)         # <class 'bool'>
type(1 < 2)        # <class 'bool'>
type(None)         # <class 'NoneType'>
```

Python supports the conversion of types using a function with the type name as `int()` or `str()`. When the conversion is not possible, Python throws an error:

```
int('12')          # 12
str(12)            # '12'
int('12.0')        # ValueError
int(alphabet)      # ValueError
int(True)          # 1
int(False)         # 0
bool(7)            # True
bool(0)            # False
bool(None)         # False
```

Like in other programming languages, the Boolean `True` and `False` values have synonyms in the other types:

False: `int`: 0, `float`: 0.0, in the none type, `None`. The empty data structures in general are synonyms of False as the empty string (`str`) `''` and the empty list, `[]`;
True: The rest.

2.6 Data Structures

2.6.1 Lists

Lists in Python are data structures that can hold any number of elements of any type. Like in strings, each element has a position, where we can read data using the position index. We can also write data to a specific index and a list grows or shrinks automatically when elements are appended, inserted, or deleted. Python manages the memory without any intervention from the programmer.

Here are some examples of lists:

```
list1 = []         # An empty list
list1 = list()     # Another way to create an empty list
list2 = [1, 2, 3]  # List containing 1, 2, and 3
```

Reading or writing a value to a position of the list is done using its index between square brackets starting from 0. If an element is read or assigned to a position that does not exist, Python returns an index error:

```
list2[1]           # 2
list2[1] = 8
list2              # [1, 8, 3]
list2[4]           # Index error
```

Lists can contain elements of different types:

```
var1 = 3.14
var2 = 'my string'
list3 = [1, var1, 'Prolog', var2]
list3              # [1, 3.14, 'Prolog', 'my string']
```

As with strings, we can extract sublists from a list using slices. The syntax is the same, but unlike strings, we can also assign a list to a slice:

```
list3[1:3]        # [3.14, 'Prolog']
list3[1:3] = [2.72, 'Perl', 'Python']
list3             # [1, 2.72, 'Perl', 'Python', 'my string']
```

We can create lists of lists:

```
list4 = [list2, list3]
    # [[1, 8, 3], [1, 2.72, 'Perl', 'Python', 'my string']]
```

where we access the elements of the inner lists with a sequence of indices between square brackets:

```
list4[0][1]     # 8
list4[1][3]     # 'Python'
```

We can also assign complete list to a variable and a list to a list of variables as in:

```
list5 = list2
[v1, v2, v3] = list5
```

where `list5` contains a copy of `list2`, and `v1`, `v2`, `v3` contain, respectively, 1, 8, and 3.

2.6.2 Built-in List Operations and Functions

Lists have built-in operators and functions. Like for strings, we can use the `+` and `*` operators to concatenate and repeat lists:

```
list2                    # [1, 8, 3]
list3[:-1]               # [1, 2.72, 'Perl', 'Python']
[1, 2, 3] + ['a', 'b']   # [1, 2, 3, 'a', 'b']
list2[:2] + list3[2:-1]  # [1, 8, 'Perl', 'Python']
list2 * 2                # [1, 8, 3, 1, 8, 3]
[0.0] * 4                # Initializes a list of four 0.0s
                         # [0.0, 0.0, 0.0, 0.0]
```

In addition to operators, lists have functions that include:

- `list.extend(elements)` that extends the list with the elements of `elements` passed as argument;
- `list.append(element)` that appends `element` to the end of the list;
- `list.insert(idx, element)` that inserts `element` at index `idx`;
- `list.remove(value)` that removes the first occurrence of value;
- `list.pop(i)`, that removes the element at index `i` and returns its value; If there is no index, `list.pop()` takes the last element in the list;
- `del list[i]`, a statement that also removes the element at index `i`. In addition, `del` can remove slices, clear the whole list, or delete the `list` variable;
- `len()`, a function that returns the length of `list`;
- `list.sort()` that sorts the list;
- `sorted()` a function that returns a sorted list.

A few examples:

```
list2                       # [1, 8, 3]
list2[1] = 2                # [1, 2, 3]
len(list2)                  # 3
list2.extend([4, 5])        # [1, 2, 3, 4, 5]
list2.append(6)             # [1, 2, 3, 4, 5, 6]
list2.append([7, 8])        # [1, 2, 3, 4, 5, 6, [7, 8]]
list2.pop(-1)               # [1, 2, 3, 4, 5, 6]
list2.remove(1)             # [2, 3, 4, 5, 6]
list2.insert(0, 'a')        # ['a', 2, 3, 4, 5, 6]
```

To know all the functions associated with a type, we can use `dir()`, as in:

```
dir(list)
```

or

```
dir(str)
```

To have help on a specific type or function, we can use `help` as in:

```
help(list)
```

and

```
help(list.append)
```

or read the online documentation.

2.6.3 Tuples

Tuples are sequences enclosed in parentheses. They are very similar to lists, except that they are immutable. Once created, we access the elements of a tuple, including slices, using the same notation as with the lists.

```
tuple1 = ()         # An empty tuple
tuple1 = tuple()    # Another way to create an empty tuple
tuple2 = (1, 2, 3, 4)
tuple2[3]           # 4
tuple2[1:4]         # (2, 3, 4)
tuple2[3] = 8       # Type error: Tuples are immutable
```

Parentheses enclosing one item could be ambiguous as `(1)`, for example, as it already denotes an arithmetic expression. That is why tuples of one item require a trailing comma:

```
type((1))   # <class 'int'>
            # Arithmetic expression corresponding to integer 1
type((1,))  # <class 'tuple'>
            # A tuple consisting of one item: integer 1
```

We can convert lists to tuples and tuples to lists:

```
list6 = ['a', 'b', 'c']
tuple3 = tuple(list6) # conversion to a tuple: ('a', 'b', 'c')
type(tuple3)          # <class 'tuple'>
list7 = list(tuple2)  # [1, 2, 3, 4]
tuple([1])            # (1,)
                      # conversion to a tuple of one item
list((1,))            # [1]
                      # conversion to a list of one item
```

Tuple can include elements of different types. If an inner element is mutable, we can change its value as in:

```
tuple4 = (tuple2, list6) # ((1, 2, 3, 4), ['a', 'b', 'c'])
tuple4[0]                # (1, 2, 3, 4),
tuple4[1]                # ['a', 'b', 'c']
tuple4[0][2]             # 3
tuple4[1][1]             # 'b'
tuple4[1][1] = 'β'       # ((1, 2, 3, 4), ['a', 'β', 'c'])
```

2.6.4 Sets

Sets are collections that have no duplicates. We create a set with a sequence enclosed in curly braces or an empty set with the `set()` function. We can then add and remove elements with the `add()` and `remove()` functions:

```
set1 = set()                      # An empty set
set2 = {'a', 'b', 'c', 'c', 'b'}  # {'a', 'b', 'c'}
set2.add('d')                     # {'a', 'b', 'c', 'd'}
set2.remove('a')                  # {'b', 'c', 'd'}
```

Sets are useful to extract the unique elements of lists or strings as in:

```
list8 = ['a', 'b', 'c', 'c', 'b']
set3 = set(list8)                # {'a', 'b', 'c'}
iliad_chars = set(iliad.lower())
        # The set of unique characters of the iliad string
```

Sets are unordered. We can create a sorted list of them using `sorted()` as in:

```
>>> sorted(iliad_chars)
['\n', ' ', '"', ',', '.', 'a', 'b', 'c', 'd', 'e', 'f',
  'g', 'h', 'i', 'l', 'n', 'o', 'p', 'r', 's', 't', 'u']
```

2.6.5 Built-in Set Functions

The set library includes the classical set operations:

- `set1.intersection(set2, ...)`
- `set1.union(set2, ...)`
- `set1.difference(set2, ...)`
- `set1.symmetric_difference(set2)`
- `set1.issuperset(set2)`
- `set1.issubset(set2)`

A few examples:

```
set2.intersection(set3)          # {'c', 'b'}
set2.union(set3)                 # {'d', 'b', 'a', 'c'}
set2.symmetric_difference(set3)  # {'a', 'd'}
set2.issubset(set3)              # False
iliad_chars.intersection(set(alphabet))
      # characters of the iliad string that are letters:
      # {'a', 's', 'g', 'p', 'u', 'h', 'c', 'l', 'i',
      #  'd', 'o', 'e', 'b', 't', 'f', 'r', 'n'}
```

2.6.6 Dictionaries

Dictionaries are collections, where the values are indexed by keys instead of ordered positions, like in lists or tuples. Counting the words of a text is a very frequent operation in natural language processing, as we will see in the rest of this book. Dictionaries are the most appropriate data structures to carry this out, where we use the keys to store the words and the values to store the counts.

We create a dictionary by assigning it a set of initial key-value pairs, possibly empty, where keys and values are separated by a colon, and then adding keys and values using the same syntax as with the lists. The statements:

```
wordcount = {}          # We create an empty dictionary
wordcount = dict()      # Another way to create a dictionary
wordcount['a'] = 21     # The key 'a' has value 21
wordcount['And'] = 10   # 'And' has value 10
wordcount['the'] = 18
```

create the dictionary `wordcount` and add three keys: `a`, `And`, `the`, whose values are 21, 10, and 18. We refer to the whole dictionary using the notation `wordcount`.

```
>>> wordcount
{'the': 18, 'a': 21, 'And': 10}
```

The order of the keys is not defined at run-time and we cannot rely on it.

The values of the resulting dictionary can be accessed by their keys with the same syntax as with lists:

```
wordcount['a']        # 21
wordcount['And']      # 10
```

A dictionary entry is created when a value is assigned to it. Its existence can be tested using the `in` Boolean function:

```
'And' in wordcount    # True
'is' in wordcount     # False
```

Just like indices for lists, the key must exist to access it, otherwise it generates an error:

```
wordcount['is']       # Key error
```

To access a key in a dictionary without risking an error, we can use the `get()` function that has a default value if the key is undefined:

- `get('And')` returns the value of the key or `None` if undefined;
- `get('is', val)` returns the value of the key or `val` if undefined.

as in:

```
wordcount.get('And')   # 10
wordcount.get('is', 0) # 0
wordcount.get('is')    # None
```

Keys can be strings, numbers, or immutable structures. Mutable keys, like a list, will generate an error:

```
my_dict = {}
my_dict[('And', 'the')] = 3  # OK, we use a tuple
my_dict[['And', 'the']] = 3  # Type error:
                             # unhashable type: 'list'
```

2.6.7 Built-in Dictionary Functions

Dictionaries have a set of built-in functions. The most useful ones are:

- `keys()` returns the keys of a dictionary;
- `values()` returns the values of a dictionary;
- `items()` returns the key-value pairs of a dictionary.

A few examples:

```
wordcount.keys()       # dict_keys(['the', 'a', 'And'])
wordcount.values()     # dict_values([18, 21, 10])
wordcount.items()      # dict_items([('the', 18), ('a', 21),
                       # ('And', 10)])
```

2.6.8 Counting the Letters of a Text

Let us finish with a program that counts the letters of a text. We use the `for in` statement to scan the `iliad` text set in lowercase letters; we increment the frequency of the current letter if it is in the dictionary or we set it to 1, if we have not seen it before.

The complete program is:

```
letter_count = {}
for letter in iliad.lower():
    if letter in alphabet:
        if letter in letter_count:
            letter_count[letter] += 1
        else:
            letter_count[letter] = 1
```

resulting in:

```
>>> letter_count
{'g': 4, 's': 10, 'o': 8, 'u': 4, 'h': 6, 'c': 3, 'l': 6,
'a': 6, 't': 6, 'd': 2, 'e': 9, 'b': 1, 'p': 2, 'f': 2,
'r': 2, 'n': 6, 'i': 3}
```

To print the result in alphabetical order, we extract the keys; we sort them; and we print the key-value pairs. We do all this with this loop:

```
for letter in sorted(letter_count.keys()):
    print(letter, letter_count[letter])
```

Running it results in:

```
a 6
b 1
c 3
d 2
e 9
...
```

By default, `sorted()` sorts the elements alphabetically. If we want to sort the letters by frequency, we can use the `key` argument of `sorted()`. `key` specifies a function whose result is used to compare the elements. In our case, we want to compare the frequencies, that is the values of the dictionary. We saw that we extract these

values with the `get` method, here `letter_count.get`, and we hence assign it to `key`.

Using `get`, the letters will be sorted from the least frequent to the most frequent. If we want to reverse this order, we use the third argument, `reverse`, a Boolean value, that we set to `True`.

Finally, our new loop:

```
for letter in sorted(letter_count.keys(),
                     key=letter_count.get, reverse=True):
    print(letter, letter_count[letter])
```

produces this output:

```
s 10
e 9
o 8
t 6
h 6
...
```

2.7 Control Structures

In Python, the control flow statements include conditionals, loops, exceptions, and functions. These statements consist of two parts, the header and the suite. The header starts with a keyword like `if`, `for`, or `while` and ends with a colon. The suite consists of the statement sequence controlled by the header; we have seen that the statement in the suite must be indented with four characters.

At this point, we may wonder how we can break expressions in multiple lines, for instance to improve the readability of a long list or long arithmetic operations. The answer is to make use of parentheses, square or curly brackets. A statement inside parentheses or brackets is equivalent to a unique logical line, even if it contains line breaks.

2.7.1 Conditionals

Python expresses conditions with the `if`, `elif`, and `else` statements as in:

```
digits = '0123456789'
punctuation = '.,;:?!'

char = '.'
if char in alphabet:
    print('Letter')
elif char in digits:
    print('Number')
elif char in punctuation:
```

```
    print('Punctuation')
else:
    print('Other')
```

that will print `Punctuation`.

2.7.2 The `for` Loop

A `for in` loop in Python iterates over the elements of a sequence such as a string or a list. This differs from languages like Perl, C or Java, where the typical `for` iteration is over numbers. If we need to create such loops, Python has the `range(start, stop, step)` function that returns a sequence of numbers. Only one argument is required: `stop`. The variables `start` and `step` will default to 0 and 1.

The next program generates the integers from 0 to 99 and computes their sum:

```
sum = 0
for i in range(100):
    sum += i
print(sum)   # Sum of integers from 0 to 99: 4950
             # Using the built-in sum() function,
             # sum(range(100)) would produce the same result.
```

The `range()` behavior is comparable to that of a list, but as a list will grow with its length, `range()` will use a constant memory. Nonetheless, we can convert a range into a list:

```
list10 = list(range(5))      # [0, 1, 2, 3, 4]
```

We have seen how to iterate over a list and over indices using `range()`. Should we want to iterate over both, we can use the `enumerate()` function. `enumerate()` takes a sequence as argument and returns a sequence of `(index, element)` pairs, where `element` is an element of the sequence and `index`, its index.

We can use `enumerate()` to get the letters of the alphabet and their index with the program:

```
for idx, letter in enumerate(alphabet):
    print(idx, letter)
```

that prints:

```
0 a
1 b
2 c
3 d
4 e
5 f
...
```

Note the parallel assignment of `idx` and `letter`.

2.7.3 The `while` Loop

The `while` loop is an alternative to `for`, although less frequent in Python programs. This loop executes a block of statements as long as a condition is true. We can reformulate the counting `for` loop in Sect 2.7.2 using `while`:

```
sum, i = 0, 0
while i < 100:
    sum += i
    i += 1
print(sum)
```

Another possible structure is to use an infinite loop and a `break` statement to exit the loop:

```
sum, i = 0, 0
while True:
    sum += i
    i += 1
    if i >= 100:
        break
print(sum)
```

Note that it is not possible to assign a variable in the condition of a `while` statement.

2.7.4 Exceptions

Python has a mechanism to handle errors so that they do not stop a program. It uses the `try` and `except` keywords. We saw in Sect. 2.5 that the conversion of the `alphabet` and `'12.0'` strings into integers prints an error and exits the program. We can handle it safely with the `try`/`except` construct:

```
try:
    int(alphabet)
    int('12.0')
except:
    pass
print('Cleared the exception!')
```

where `pass` is an empty statement serving as a placeholder for the `except` block.

It is also possible, and better, to tell `except` to catch specific exceptions as in:

```
try:
    int(alphabet)
    int('12.0')
except ValueError:
    print('Caught a value error!')
```

```
except TypeError:
    print('Caught a type error!')
```

that prints:

```
Caught a value error!
```

2.8 Functions

We define a function in Python with the `def` keyword and we use `return` to return the results. In Sect. 2.6.6, we wrote a small program to count the letters of a text. Let us create a function from it that accepts any text instead of `iliad`. We also add a Boolean, `lc`, to set the text in lowercase:

```
def count_letters(text, lc=True):
    letter_count = {}
    if lc:
        text = text.lower()
    for letter in text:
        if letter.lower() in alphabet:
            if letter in letter_count:
                letter_count[letter] += 1
            else:
                letter_count[letter] = 1
    return letter_count
```

We call the function with the two parameters:

```
count_letters(iliad, True)
```

If most of the calls use a parameter with a specific value, we can use it as default with the notation:

```
def count_letters(text, lc=True):
```

In this case, the call `count_letters(iliad)` with one single parameter will be equivalent to:

```
count_letters(iliad, True)
```

2.9 Comprehensions and Generators

2.9.1 Comprehensions

Instead of loops, the comprehensions are an alternative, concise syntactic notation to to create lists, sets, or dictionaries.

An example of elegant use of list comprehensions is given by Norvig (2007), who wrote a delightful spelling corrector in 21 lines of Python. To verify that a word is correctly written, spell checkers look it up in a dictionary. If a word is not in the dictionary, and is presumably a typo, spelling correctors generate candidate corrections through, for instance, the deletion of one character of this word, in the hope that it can find a match in the dictionary. See Sect. 3.5 of this book for details.

Given an input word, we can generate all the one-character deletions in two steps: First, we split the word into two parts; then we delete the first letter of the second part. We can write this operation in two comprehensions, whose syntax is close to the set comprehension in set theory. First, we generate the splits:

```
splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
```

where we iterate over the sequence of character indices and we create pairs consisting of a prefix and a rest.

If the input word is *acress*, the resulting list in `splits` is:

```
[('', 'acress'), ('a', 'cress'), ('ac', 'ress'),
('acr', 'ess'), ('acre', 'ss'), ('acres', 's'), ('acress', '')]
```

Then, we apply the deletions, where we concatenate the prefix and the rest deprived from its first character. We check that the rest is not an empty list:

```
deletes = [a + b[1:] for a, b in splits if b]
```

The result in `deletes` is:

```
['cress', 'aress', 'acess', 'acrss', 'acres', 'acres']
```

where *cress* and *acres* are dictionary words.

The comprehensions are equivalent to loops. The first one to:

```
splits = []
for i in range(len(word) + 1):
    splits.append((word[:i], word[i:]))
```

and the second one:

```
deletes = []
for a, b in splits:
    if b:
        deletes.append(a + b[1:])
```

We can create set and dictionary comprehensions the same way by replacing the enclosing square brackets with curly braces: `{}`.

2.9.2 Generators

List comprehensions are stored in memory. If the list is large, it can exceed the computer capacity. Generators generate the elements on demand instead and can handle much longer sequences.

Generators have a syntax that is identical to the list comprehensions except that we replace the square brackets with parentheses:

```
splits_generator = ((word[:i], word[i:])
                    for i in range(len(word) + 1))
```

We can iterate over this generator exactly as with a list. The statement:

```
for i in splits_generator: print(i)
```

prints

```
('', 'acress')
('a', 'cress')
('ac', 'ress')
('acr', 'ess')
('acre', 'ss')
('acres', 's')
('acress', '')
```

However, this iteration can only be done once. We need to create the generator again to retraverse the sequence.

Finally, we can also use functions to create generators. We replace the `return` keyword with `yield` to do this, as in the function:

```
def splits_generator_function():
    for i in range(len(word) + 1):
        yield (word[:i], word[i:])
```

that returns a generator identical to the previous one:

```
splits_generator = splits_generator_function()
```

2.9.3 Iterators

We just saw that we can iterate only once over a generator. Objects with this property in Python are called iterators. Iterators are very efficient devices and, at the same time, probably less intuitive than lists for beginners.

Let us give some examples with a useful iterator: `zip()`. Let us first create three strings with the Latin, Greek, and Russian Cyrillic alphabets:

```
latin_alphabet = 'abcdefghijklmnopqrstuvwxyz'
len(latin_alphabet)          # 26
greek_alphabet = 'αβγδεζηθικλμνξοπρστυφχψω'
len(greek_alphabet)          # 24
cyrillic_alphabet = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
len(cyrillic_alphabet)       # 33
```

`zip()` weaves strings, lists, or tuples and creates an iterator of tuples, where each tuple contains the elements with the same index: `latin_alphabet[0]` and `greek_alphabet[0]`, `latin_alphabet[1]` and `greek_alphabet[1]`, and so on. If the strings are of different sizes, `zip()` will stop at the shortest.

The following code applies `zip()` to the three first letters of our alphabets:

```
la_gr = zip(latin_alphabet[:3], greek_alphabet[:3])
la_gr_cy = zip(latin_alphabet[:3], greek_alphabet[:3],
               cyrillic_alphabet[0:3])
```

and creates two iterators with the tuples:

```
la_gr # ('a', 'α'), ('b', 'β'), ('c', 'γ')
la_gr_cy # ('a', 'α', 'а'), ('b', 'β', 'б'), ('c', 'γ', 'в')
```

Once created, we access the elements of an iterator with `__next()__` as in:

```
la_gr.__next__()   # ('a', 'α')
la_gr.__next__()   # ('b', 'β')
la_gr.__next__()   # ('c', 'γ')
```

When we reach the end and there are no more elements, Python raises an exception:

```
la_gr.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

If we want to use this iterator again, we have to recreate it.

Another way to traverse this sequence multiple times is to convert the iterator to a list as in:

```
la_gr_cy_list = list(la_gr_cy)
la_gr_cy_list
       # [('a', 'α', 'а'), ('b', 'β', 'б'), ('c', 'γ', 'в')]
la_gr_cy_list
       # [('a', 'α', 'а'), ('b', 'β', 'б'), ('c', 'γ', 'в')]
```

We must be aware that the list conversion runs the iterator through the sequence and if we try to convert `la_gr_cy` a second time, we just get an empty list:

```
la_gr_cy_list = list(la_gr_cy)
la_gr_cy_list  # []
```

To restore the original lists of alphabet, we can use the `zip(*)` inverse function:

```
la_gr_cy = zip(latin_alphabet[:3], greek_alphabet[:3],
               cyrillic_alphabet[0:3])
       # ('a', 'α', 'а'), ('b', 'β', 'б'), ('c', 'γ', 'в')
zip(*la_gr_cy)
       # ('a', 'b', 'c'), ('α', 'β', 'γ'), ('а', 'б', 'в')
```

Finally, we can convert lists to iterators using `iter()`.

2.10 Modules

Python comes with a very large set of libraries called modules like, for example, the `math` module that contains a set of mathematical functions. We load a module with the `import` keyword and we use its functions with the module name as a prefix followed by a dot:

```
import math
math.sqrt(2)            # 1.4142135623730951
math.sin(math.pi/2)     # 1.0
math.log(8, 2)          # 3.0
```

We can create an alias name to the modules with the `as` keyword:

```
import statistics as stats
stats.mean([1, 2, 3, 4, 5])   # 3.0
stats.stdev([1, 2, 3, 4, 5])  # 1.5811388300841898
```

Modules are just files, whose names are the module names with the `.py` suffix. To import a file, Python searches first the standard library, the files in the current folder, and then the files in `PYTHONPATH`.

When Python imports a module, it executes its statements just as when we run:

```
$ python module.py
```

If we want to have a different execution when we run the program from the command line and when we import it, we need to include this condition:

```
if __name__ == '__main__':
    print("Running the program")
    # Other statements
else:
    print("Importing the program")
    # Other statements
```

The first member is executed when the program is run from the command line and the second one, when we import it.

2.11 Installing Modules

Python comes with a standard library of modules like `math`. Although comprehensive, we will use external libraries in the next chapters that are not part of the standard release as the `regex` module in Chap. 4. We can use `pip`, the Python package manager to install the modules we need. `pip` will retrieve them from the Python package index (PyPI) and fetch them for us.

To install `regex`, we just run the command:

```
$ pip install regex
```

or

```
$ python -m pip install regex
```

and if we want to upgrade an already installed module, we run:

```
$ python -m pip install --upgrade regex
```

Another option is to use a Python distribution with pre-installed packages like Anaconda (`https://www.continuum.io/downloads`). Nonetheless, even if Anaconda has many packages, it does not include `regex` and we will have to install it.

2.12 Basic File Input/Output

Python has a set of built-in input/output functions to read and write files: `open()`, `read()`, `write()`, and `close()`.

The next lines open and read the `iliad.txt` file, count the characters, and write the results in the `iliad_stats.txt` file:

```
f_iliad = open('iliad.txt', 'r')          # open a file
iliad_txt = f_iliad.read()                # read all the file
f_iliad.close()                           # close the file
iliad_stats = count_letters(iliad_txt)  # count the letters
with open('iliad_stats.txt', 'w') as f:
    f.write(str(iliad_stats))
    # we automatically close the file
```

where `open()` opens a file in the read-only mode, `r`, and returns a file object; `read()` reads the entire content of the file and returns a string; `close()` closes the file object; `count_letter()` counts the letters; and finally the `with` statement is a shorthand to handle exceptions and close the file automatically after the block: `open()` creates a new file using the write mode, `w`, and `write()` writes the results as a string.

In addition to these base functions, Python has modules to read and write a large variety of file formats.

2.13 Memo Functions and Decorators

2.13.1 Memo Functions

Memo functions are functions that remember a result instead of computing it. This process is also called *memoization*. The Fibonacci series is a case, where memo functions provide a dramatic execution speed up.

The Fibonacci sequence is defined by the relation:

$$F(n) = F(n-1) + F(n-2)$$

with $F(1) = F(2) = 1$.

A naïve implementation in Python is straightforward:

```
def fibonacci(n):
    if n == 1: return 1
    elif n == 2: return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

however, this function has an expensive double recursion that we can drastically improve by storing the results in a dictionary. This store, `f_numbers`, will save an exponential number of recalculations:

```
f_numbers = {}

def fibonacci2(n):
    if n == 1: return 1
    elif n == 2: return 1
    elif n in f_numbers:
        return f_numbers[n]
    else:
        f_numbers[n] = fibonacci2(n - 1) + fibonacci2(n - 2)
        return f_numbers[n]
```

2.13.2 Decorators

Python decorators are syntactic notations to simplify the writing of memo functions (they can be used for other purposes too).

Decorators need a generic memo function to cache the results already computed. Let us define it:

```
def memo_function(f):
    cache = {}

    def memo(x):
        if x in cache:
            return cache[x]
        else:
            cache[x] = f(x)
            return cache[x]

    return memo
```

Using this memo function, we can redefine `fibonacci()` with the statement:

```
fibonacci = memo_function(fibonacci)
```

that results in `memo()` being assigned to the `fibonacci()` function. When we call `fibonacci()`, we in fact call `memo()` that will either lookup the cache or call the original `fibonacci()` function.

One detail may be puzzling: How does the new function know of the `cache()` variable and its initialization as well as the value of the `f` argument, the original `fibonacci()` function? This is because Python implements a closure mechanism that gives the inner functions access to the local variables of their enclosing function.

Now the decorators: Python provides a short notation for memo functions; instead of writing:

```
fibonacci = memo_function(fibonacci)
```

we just decorate `fibonacci()` with the `@memo_function` line before it:

```
@memo_function
def fibonacci(n):
...
```

2.14 Classes and Objects

Although not obvious at first sight, Python is an object-oriented language, where all the language entities are objects inheriting from a class: The `str` class for the strings, for instance. Each class has a set of methods that we call with the `object.method()` notation.

We define our own classes with the `class` keyword. In Sect. 2.8, we wrote a `count_letters()` function that basically is to be applied to a text. Let us reflect this with a `Text` class and let us encapsulate this function as a method in this class. In addition, we give the `Text` class four variables: The text, its length, and the letter counts, which will be specific to each object and the `alphabet` string that will be shared by all the objects. We say that `alphabet` is a class variable while `text`, `length`, and `letter_count` are instance variables.

We encapsulate a function by inserting it as a block inside the class. Among the methods, one of them, the **constructor**, is called at the creation of an object. It has the `__init()__` name. This notation in Python is, unfortunately, not as intuitive as the rest of the language, and we need to add a `self` extra-parameter to the methods as well as to the instance variables. This `self` keyword denotes the object itself. We use `__init()__` to assign an initial value to the `text`, `length`, and `letter_count` variables.

Finally, we have the class:

```
class Text:
    """Text class to hold and process text"""

    alphabet = 'abcdefghijklmnopqrstuvwxyz'

    def __init__(self, text=None):
        """The constructor called when an object
        is created"""
```

```
        self.text = text
        self.length = len(text)
        self.letter_count = {}

    def count_letters(self, lc=True):
        """Function to count the letters of a text"""

        letter_count = {}
        if lc:
            text = self.text.lower()
        else:
            text = self.text
        for letter in text:
            if letter.lower() in self.alphabet:
                if letter in letter_count:
                    letter_count[letter] += 1
                else:
                    letter_count[letter] = 1
        self.letter_count = letter_count
        return letter_count
```

We create new objects using the `Text(init_value)` syntax:

```
txt = Text("""Tell me, O Muse, of that many-sided hero who
traveled far and wide after he had sacked the famous town
of Troy.""")
```

A class has its own type:

```
type(txt)           # <class '__main__.Text'>
```

We access the instance variables using this notation:

```
text.length         # 111
```

We create and assign new instance variables the same way:

```
txt.my_var = 'a'    # a new instance variable with value 'a'
txt.text = open('iliad.txt', 'r').read()
                    # txt.text is now the content of the file
```

and we call methods with the same notation:

```
txt.count_letters() # return the letter counts of txt.text
```

Finally, we added short descriptions of the class and its methods in the form of **docstrings**: Strings being the first statement of the class, method, or function. Docstrings are very useful to document a program. We access them using the `.__doc__` variable as in:

```
Text.__doc__    # 'Text class to hold and process text'
Text.count_letters.__doc__
                # 'Function to count the letters of a text'
```

or with the `help()` function.

2.15 Functional Programming

Python provides some functional programming mechanisms with map and reduce functions.

2.15.1 `map()`

`map()` enables us to apply a function to all the elements of an iterable, a list for instance. The first argument of `map()` is the function to apply and the second one, the iterable. `map()` returns an iterator.

Let us use `map()` to compute the length of a sequence of texts, in our case, the first sentences of the *Iliad* and the *Odyssey*. We apply `len()` to the list of strings and we convert the resulting iterator to a list to print it.

```
odyssey = """Tell me, O Muse, of that many-sided hero who
traveled far and wide after he had sacked the famous town
of Troy."""

text_lengths = map(len, [iliad, odyssey])
list(text_lengths)     # [100, 111]
```

2.15.2 Lambda Expressions

Let us now suppose that we have a list of files instead of strings, here `iliad.txt` and `odyssey.txt`. To deal with this list, we can replace `len()` in `map()` with a function that reads a file and computes its length:

```
def file_length(file):
    return len(open(file).read())
```

For such a short function, a lambda expression can do the job more compactly. A lambda is an anonymous function, denoted with the lambda keyword, followed by the function parameters, a colon, and the returned expression. To compute the length of a file, we write the lambda:

```
lambda file: len(open(file).read())
```

and we apply it to our list of files:

```
files = ['iliad.txt', 'odyssey.txt']
text_lengths = map(lambda x: len(open(x).read()), files)
list(text_lengths)                    # [809768, 611742]
```

We can return multiple values using tuples. If we want to both keep the text and its length in the form of a pair: $(text, length)$, we just write:

```
text_lengths = (
    map(lambda x: (open(x).read(), len(open(x).read())),
        files))
text_lengths = list(text_lengths)
[text_lengths[0][1], text_lengths[1][1]]  # [809768, 611742]
```

In the previous piece of code, we had to read the text twice: In the first element of the pair and in the second one. We can use two `map()` calls instead: One to read the files and a second to compute the lengths. This results in:

```
text_lengths = (
    map(lambda x: (x, len(x)),
        map(lambda x: open(x).read(), files)))
text_lengths = list(text_lengths)
[text_lengths[0][1], text_lengths[1][1]]  # [809768, 611742]
```

2.15.3 `reduce()`

`reduce()` is a complement to `map()` that applies an operation to pairs of elements of a sequence. We can use `reduce()` and the addition to compute the total number of characters of our set of files. We formulate it as a lambda expression:

```
lambda x, y: x[1] + y[1]
```

to sum the consecutive elements, where the length of each file is the second element in the pair; the first one being the text.

`reduce()` is part of the `functools` module and we have to import it. The resulting code is:

```
import functools

char_count = functools.reduce(
    lambda x, y: x[1] + y[1],
    map(lambda x: (x, len(x)),
        map(lambda x: open(x).read(), files)))
char_count     # 1421510
```

2.15.4 `filter()`

`filter()` is a third function that we can use to keep the elements of an iterable that satisfy a condition. `filter()` has two arguments: A function, possibly a lambda, and an iterable. It returns the elements of the iterable for which the function is true.

As an example of the `filter()` function, let us write a piece of code to extract and count the lowercase vowels of a text.

We need first a lambda that returns true if a character `x` is a vowel:

```
lambda x : x in 'aeiou'
```

that we apply to the `iliad` string to obtain all its vowels:

```
''.join(filter(lambda x : x in 'aeiou', iliad))
     # ioeeaeoieooeeuaououeiuoeaea
```

We can apply the same code to a whole file:

```
''.join(filter(lambda x: x in 'aeiou',
               open('iliad.txt').read()))
```

and easily extend the extraction to a list of files using `map()`:

```
map(lambda y:
    ''.join(filter(lambda x: x in 'aeiou',
                   open(y).read())),
    files)
```

We finally count the vowels in the two files using `len()` that we apply with a second `map()`:

```
list(map(len,
         map(lambda y:
             ''.join(filter(lambda x: x in 'aeiou',
                            open(y).read())),
             files)))  # [231874, 176190]
```

2.16 Further Reading

Python has become very popular and there are plenty of good books or tutorials to complement this introduction. `Python.org` is the official site of the Python software foundation, where one can find the latest Python releases, documentation, tutorials, news, etc. It also contains masses of pointers to Python resources. Anaconda is an alternative Python distribution that includes many libraries (`www.continuum.io/downloads`).

Python comes with a integrated development environment (IDE) called IDLE that fulfills basic needs. PyCharm is a more elaborate code editor with a beautiful interface. It has a free community edition (`www.jetbrains.com/pycharm/`). IPython is an interactive computing platform, where the programmer can mix code and text in the form of notebooks.

Among all the computer science publishers, O'Reilly Media has an impressive collection of books on Python that ranges from introductions to very detailed or

domain-oriented monographs. Payne (2015) and Lutz (2013) are two examples of this variety.

Finally, online course providers, like Coursera or edX, offer free and high-quality Python courses from top universities and open to anyone.

3

Corpus Processing Tools

A. a. a.	Je.	I.a.	[A, a, a,] domine deus, ecce nescio loqui.
		XIIII.b.	[A, a, a, domine deus,] prophete dicunt eis.
	Eze.	IIII.d.	[A, a, a,] domine deus ecce anima mea non est
		XXI.a.	[A, a, a,] domine deus.
	Joel	I.c.	[A, a, a,] diei.
Aaron	exo.	IIII.c	[Aaron...] egredietur in occursum
		VII.a.	[Aaron frater tuus] erit propheta tuus.
		XVII.d.	[Aaron autem et] Hur sustentabant manus.
		XXIIII.d.	habetis Aaron et Hur vobiscum.
	...		

First lines from the third concordance to the Vulgate.
Abbreviations are spelled out for clarity.
Bibliothèque nationale de France. Manuscrit latin 515. Thirteenth century.

3.1 Corpora

A corpus, plural corpora, is a collection of texts or speech stored in an electronic machine-readable format. A few years ago, large electronic corpora of more than a million of words were rare, expensive, or simply not available. At present, huge quantities of texts are accessible in many languages of the world. They can easily be collected from a variety of sources, most notably the Internet, where corpora of hundreds of millions of words are within the reach of most computational linguists.

3.1.1 Types of Corpora

Some corpora focus on specific genres: law, science, novels, news broadcasts, electronic correspondence, or transcriptions of telephone calls or conversations. Others try to gather a wider variety of running texts. Texts collected from a unique source, say from scientific magazines, will probably be slanted toward some specific words that do not appear in everyday life. Table 3.1 compares the most frequent words in

Table 3.1. List of the most frequent words in present texts and in the book of Genesis. After Crystal (1997)

	English	**French**	**German**
Most frequent words in a collection of contemporary running texts	*the*	*de*	*der*
	of	*le* (article)	*die*
	to	*la* (article)	*und*
	in	*et*	*in*
	and	*les*	*des*
Most frequent words in Genesis	*and*	*et*	*und*
	the	*de*	*die*
	of	*la*	*der*
	his	*à*	*da*
	he	*il*	*er*

the book of Genesis and in a collection of contemporary running texts. It gives an example of such a discrepancy. The choice of documents to include in a corpus must then be varied to survey comprehensively and accurately a language usage. This process is referred to as balancing a corpus.

Balancing a corpus is a difficult and costly task. It requires collecting data from a wide range of sources: fiction, newspapers, technical, and popular literature. Balanced corpora extend to spoken data. The Linguistic Data Consortium (LDC) from the University of Pennsylvania and the European Language Resources Association (ELRA), among other organizations, distribute written and spoken corpus collections. They feature samples of magazines, laws, parallel texts in English, French, German, Spanish, Chinese, Arabic, telephone calls, radio broadcasts, etc.

In addition to raw texts, some corpora are annotated. Each of their words is labeled with a linguistic tag such as a part of speech or a semantic category. The annotation is done either manually or semiautomatically. Spoken corpora contain the transcription of spoken conversations. This transcription may be aligned with the speech signal and sometimes includes prosodic annotation: pause, stress, etc. Annotation tags, paragraph and sentence boundaries, parts of speech, syntactic or semantic categories follow a variety of standards, which are called markup languages.

Among annotated corpora, treebanks deserve a specific mention. They are collections of parse trees or more generally syntactic structures of sentences. The production of a treebank generally requires a team of linguists to parenthesize the constituents of a corpus or to arrange them in a dependency structure. Annotated corpora require a fair amount of handwork and are therefore more expensive than raw texts. Treebanks involve even more clerical work and are relatively rare. The Penn Treebank (Marcus et al., 1993) from the University of Pennsylvania is a widely cited example for English.

A last word on annotated corpora: in tests, we will benchmark automatic methods against manual annotation, which is often called the gold standard. We will assume the hand annotation perfect, although this is not true in practice. Some errors slip into

hand-annotated corpora, even in those of the best quality, and the annotators may not agree between them. The scope of agreement varies depending on the annotation task. The inter-annotator agreement is generally high for parts of speech that are relatively well defined. It is lower when determining the sense of a word, for which annotators may have different interpretations. This inter-annotator agreement defines then a sort of upper bound of the human performance. It is a useful figure to conduct a reasonable assessment of results obtained by automatic methods as well as their potential for improvements.

3.1.2 Corpora and Lexicon Building

Lexicons and dictionaries are intended to give word lists, to provide a reader with word senses and meanings, and to outline their usage. Dictionaries' main purpose is related to lexical semantics. Lexicography is the science of building lexicons and writing dictionaries. It uses electronic corpora extensively.

The basic data of a dictionary is a word list. Such lists can be drawn manually or automatically from corpora. Then, lexicographers write the word definitions and choose citations illustrating the words. Since most of the time, current meanings are obvious to the reader, meticulous lexicographers tended to collect examples – citations – reflecting a rare usage. Computerized corpora can help lexicographers avoid this pitfall by extracting all the citations that exemplify a word. An experienced lexicographer will then select the most representative examples that reflect the language with more relevance. S/he will prefer and describe more frequent usage and possibly set aside others.

Finding a citation involves sampling a fragment of text surrounding a given word. In addition, the context of a word can be more precisely measured by finding recurrent pairs of words, or most-frequent neighbors. The first process results in concordance tables, and the second one in collocations.

A **concordance** is an alphabetical index of all the words in a text, or the most significant ones, where each word is related to a comprehensive list of passages where the word is present. Passages may start with the word or be centered on it and surrounded by a limited number of words before and after it (Table 3.2 and incipit of this chapter). Furthermore, concordances feature a system of reference to connect each passage to the book, chapter, page, paragraph, or verse, where it occurs.

Concordance tables were first produced for antiquity and religious studies. Hugh of St-Cher is known to have directed the first concordance to the scriptures in the thirteenth century. It comprised about 11,800 words ranging from *A, a, a.* to *Zorobabel* and 130,000 references (Rouse and Rouse, 1974). Other more elaborate concordances take word morphology into account or group words together into semantic themes. Sœur Jeanne d'Arc (1970) produced an example of such a concordance for Bible studies.

Concordancing is a powerful tool to study usage patterns and to write definitions. It also provides evidence on certain preferences between verbs and prepositions, adjectives and nouns, recurring expressions, or common syntactic forms. These couples are referred to as **collocations**. Church and Mercer (1993) cite a striking example of

Table 3.2. Concordance of *miracle* in the Gospel of John. English text: King James version; French text: Augustin Crampon; German text: Luther's Bible

Language	Concordances

English	`l now. This beginning of miracles did Jesus in Cana of Ga`
	`name, when they saw the miracles which he did. But Jesus`
	`for no man can do these miracles that thou doest, except`
	`This is again the second miracle that Jesus did, when he`
	`im, because they saw his miracles which he did on them th`

French	`Galilée, le premier des miracles que fit Jésus, et il ma`
	`que, beaucoup voyant les miracles qu'il faisait, crurent`
	`nne ne saurait faire les miracles que vous faites, si Die`
	`maison. Ce fut le second miracle que fit Jésus en revenan`
	`parce qu'elle voyait les miracles qu'il opérait sur ceux`

German	`alten. Das ist das erste Zeichen, das Jesus tat, geschehe`
	`as zeigst du uns für ein Zeichen, daß du dies tun darfst?`
	`seinen Namen, da sie die Zeichen sahen, die er tat. Aber`
	`n; denn niemand kann die Zeichen tun, die du tust, es sei`
	`h zu ihm: Wenn ihr nicht Zeichen und Wunder seht, so glau`

Table 3.3. Comparing *strong* and *powerful*. The German words *eng* and *schmal* 'narrow' are near-synonyms, but have different collocates

	English	French	German
You say	*Strong tea* *Powerful computer*	*Thé fort* *Ordinateur puissant*	*Schmales Gesicht* *Enge Kleidung*
You don't say	*Strong computer* *Powerful tea*	*Thé puissant* *Ordinateur fort*	*Schmale Kleidung* *Enges Gesicht*

idiosyncratic collocations of *strong* and *powerful*. While *strong* and *powerful* have similar definitions, they occur in different contexts, as shown in Table 3.3.

Table 3.4 shows additional collocations of *strong* and *powerful*. These word preferences cannot be explained using rational definitions, but can be observed in corpora. A variety of statistical tests can measure the strength of pairs, and we can extract them automatically from a corpus.

3.1.3 Corpora as Knowledge Sources for the Linguist

In the early 1990s, computer-based corpus analysis completely renewed empirical methods in linguistics. It helped design and implement many of the techniques presented in this book. As we saw with dictionaries, corpus analysis helps lexicographers acquire lexical knowledge and describe language usage. More generally, corpora enable us to experiment with tools and to confront theories and models on real data. For most language analysis programs, collecting relevant corpora of texts is

Table 3.4. Word preferences of *strong* and *powerful* collected from the Associated Press corpus. Numbers in columns indicate the number of collocation occurrences with word *w*. After Church and Mercer (1993)

Preference for *strong* over *powerful*			**Preference for *powerful* over *strong***		
strong w	*powerful w*	*w*	*strong w*	*powerful w*	*w*
161	0	*showing*	1	32	*than*
175	2	*support*	1	32	*figure*
106	0	*defense*	3	31	*minority*
...					

then a necessary step to define specifications and measure performances. Let us take the examples of part-of-speech taggers, parsers, and dialogue systems.

Annotated corpora are essential tools to develop part-of-speech taggers or parsers. A first purpose is to measure the tagging or parsing performance. The tagger or parser is run on texts and their result is compared to hand annotation, which serves as a reference. A linguist or an engineer can then determine the accuracy, the robustness of an algorithm or a parsing model and see how well it scales up by applying it to a variety of texts.

A second purpose of annotated corpora is to be a knowledge source to refine tagging techniques and improve grammars. While developing a grammar, a linguist can see if changing a rule improves or deteriorates results. The tool tuning is then done manually. Using statistical or machine-learning techniques, annotated corpora also enable researchers to create models, and identify parameters automatically or semiautomatically to tag or parse a text. We will see this in Chap. **??**.

A dialogue corpus between a user and a machine is also critical to develop an interactive spoken system. The corpus is usually collected through fake dialogues between a real user and a person simulating the machine answers. Repeating such experiments with a reasonable number of users enables us to acquire a text set covering what the machine can expect from potential users. It is then easier to determine the vocabulary of an application, to have a precise idea of word frequencies, and to know the average length of sentences. In addition, the dialogue corpus enables the analyst to understand what the user expects from the machine and how s/he interacts with it.

3.2 Regular Expressions

Regular expressions define a notation for strings of characters such as *abc*, but sets of strings such as those composed of one *a*, zero or more *b*'s, and one *c*. We can represent this set using a compact notation: `ab*c`, where the star symbol means any number of the preceding character. This notation is precisely a regular expression or regex. Regular expressions are very powerful devices to describe patterns to search in a text.

Table 3.5. Examples of simple patterns and matching results

Pattern	String
`regular`	"A section on regular expressions"
`Prolog`	"The Prolog language"
`the`	"The book of the life"

Regular expressions are composed of literal characters, that is, ordinary text characters, like `abc`, and of metacharacters, like `*`, that have a special meaning. The simplest form of regular expressions is a sequence of literal characters: letters, numbers, spaces, or punctuation signs. The regexes `regular` and `Prolog` match, respectively, the strings *regular* or *Prolog* contained in a text. Table 3.5 shows examples of pattern matching with literal characters. Regular expressions are case-sensitive and match the first instance of the string or all its instances in a text, depending on the regex language that is used.

There are currently a dozen major regular expression dialects freely available. Their common ancestor is `grep`, which stands for global/regular expression/print. `grep`, together with `egrep`, a modern version of it, is a standard Unix tool that prints out all the lines of a file that contain a given pattern. The `grep` user interface conforms to the Unix command-line style. It consists of the command name, here `grep`, options, and the arguments. The first argument is the regular expression delimited by single straight quotes. The next arguments are the files where to search the pattern:

```
grep 'regular expression' file1 file2 ... filen
```

The Unix command:

```
grep 'abc' myFile
```

prints all the lines of file `myFile` containing the string *abc* and

```
grep 'ab*c' myFile1 myFile2
```

prints all the lines of file `myFile1` and `myFile2` containing the strings *ac*, *abc*, *abbc*, *abbbc*, etc.

`grep` had a considerable influence, and most programming languages, including Perl, Python, Java, and C#, have now some support for regexes. All the regex variants – or flavors – adhere to an analog syntax, with some differences, however, that hinder a universal compatibility.

In the following sections, we will use the syntax defined by Perl. Because of its built-in support for regexes and its simplicity, Perl was immediately recognized as a real innovation in the world of scripting languages and was adopted by millions of programmers. It is probably Perl that made regular expressions a mainstream programming technique and, in return, it explains why the Perl regex syntax became a sort of *de facto* standard that inspires most modern regex flavors, including that of Python. The set of regular expressions that follows Perl is also called *Perl compatible regular expressions* (PCRE).

Table 3.6. Repetition metacharacters (quantifiers)

Metachar	Description	Example
`*`	Matches any number of occurrences of the previous character – zero or more	`ac*e` matches strings `ae`, `ace`, `acce`, `accce`, etc. as in "The aerial acceleration alerted the ace pilot"
`?`	Matches at most one occurrence of the previous character – zero or one	`ac?e` matches `ae` and `ace` as in "The aerial acceleration alerted the ace pilot"
`+`	Matches one or more occurrences of the previous character	`ac+e` matches `ace`, `acce`, `accce`, etc. as in as in "The aerial acceleration alerted the ace pilot"
`{n}`	Matches exactly *n* occurrences of the previous character	`ac{2}e` matches `acce` as in "The aerial acceleration alerted the ace pilot"
`{n,}`	Matches *n* or more occurrences of the previous character	`ac{2,}e` matches `acce`, `accce`, etc.
`{n,m}`	Matches from *n* to *m* occurrences of the previous character	`ac{2,4}e` matches `acce`, `accce`, and `acccce`.

3.2.1 Repetition Metacharacters

We saw that the metacharacter `*` expressed a repetition of zero or more characters, as in `ab*c`. Other characters that describe repetitions are the question mark, `?`, the plus, `+`, and the range quantifiers `{n,m}` matching a specified range of occurrences (Table 3.6). The star symbol is also called the closure operator or the Kleene star.

3.2.2 The Dot Metacharacter

The dot `.` is also a metacharacter that matches one occurrence of any character of the alphabet except a new line. For example, `a.e` matches the strings *ale* and *ace* in the sentence:

The aerial acceleration alerted the ace pilot

as well as *age*, *ape*, *are*, *ate*, *awe*, *axe*, or *aae*, *aAe*, *abe*, *aBe*, *a1e*, etc. We can combine the dot and the star in the expression `.*` to match any string of characters until we encounter a new line.

3.2.3 The Escape Character

If the pattern to search contains a character that is also a metacharacter, for instance, "?", we need to indicate it to the regex engine using a backslash `\` before it. We saw that `abc?` matches *ab* and *abc*. The expression `abc\?` matches the string *abc*?. In the same vein, `abc\.` matches the string *abc*., and `a\*bc` matches *a*bc*.

We call the backslash an escape character. It transforms a metacharacter into a literal symbol. We can also say that we "quote" a metacharacter with a backslash. In Python, we must use a backslash escape with the 14 following characters:

```
. ^ $ * + ? { } [ ] \ | ( )
```

to search them literally.

As a matter of fact, the backslash is not always necessary as sometimes Python can guess from the context that a character has a literal meaning. This is the case for the braces, for instance, that Python interprets as literals outside the expressions, `{n}`, `{n,}`, and `{n,m}`. Anyway, it is always safer to use a backslash escape to avoid ambiguities.

3.2.4 The Longest Match

The description of repetition metacharacters in Table 3.6 sometimes makes string matching ambiguous, as with the string `aabbc` and the regex `a+b*`, which could have six possible matches: *a*, *aa*, *ab*, *aab*, *abb*, and *aabb*. In fact, matching algorithms use two rules that are common to all the regex languages:

1. They match as early as they can in a string.
2. They match as many characters as they can.

Hence, `a+b*` matches *aabb*, which is the longest possible match. The matching strategy of repetition metacharacters is said to be greedy.

In some cases, the greedy strategy is not appropriate. To display the sentence

They match **as early** and **as many** characters as they can.

in a web page with two phrases set in bold, we need specific tags that we will insert in the source file. Using HTML, the language of the web, the sentence will probably be annotated as

```
They match <b>as early</b> and <b>as many</b> characters as
they can.
```

where `<b>` and `</b>` mark respectively the beginning and the end of a phrase set in bold. (We will see annotation frameworks in more detail in Chap. 4.)

A regular expression to search and extract phrases in bold could be:

```
<b>.*</b>
```

Unfortunately, applying this regex to the sentence will match one single string:

```
<b>as early</b> and <b>as many</b>
```

which is not what we wanted. In fact, this is not a surprise. As we saw, the regex engine matches as early as it can, i.e., from the first `<b>` and as many characters as it can up to the second `</b>`.

A possible solution is to modify the behavior of repetition metacharacters and make them "lazy." They will then consume as few characters as possible. We create the lazy variant of a repetition metacharacter by appending a question mark to it (Table 3.7). The regex

Table 3.7. Lazy metacharacters

Metachar	Description
`*?`	Matches any number of occurrences of the previous character – zero or more
`??`	Matches at most one occurrence of the previous character – zero or one
`+?`	Matches one or more occurrences of the previous character
`{n}?`	Matches exactly *n* occurrences of the previous character
`{n,}?`	Matches *n* or more occurrences of the previous character
`{n,m}?`	Matches from *n* to *m* occurrences of the previous character

```
<b>.*?</b>
```

will then match the two intended strings,

`<b>as early</b>` and `<b>as many</b>`.

3.2.5 Character Classes

We saw that the dot, `.`, represented any character of the alphabet. It is possible to define smaller subsets or **classes**. A list of characters between square brackets `[...]` matches any character contained in the list. The expression `[abc]` means one occurrence of either `a`, `b`, or `c`; `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]` means one uppercase unaccented letter; and `[0123456789]` means one digit. We can concatenate character classes, literal characters, and metacharacters, as in the expressions `[0123456789]+` and `[0123456789]+\.[0123456789]+`, that match, respectively, integers and decimal numbers.

Character classes are useful to search patterns with spelling differences, such as `[Cc]omputer [Ss]cience`, which matches four different strings:

Computer Science
Computer science
computer Science
computer science

Negated Character Classes.

We can define the complement of a character class, that is, the characters of the alphabet that are not member of the class, using the caret symbol, `^`, as the first symbol inside the square brackets. For example:

- the expression `[^a]` means any character that is not an *a*;
- `[^0123456789]` means any character that is not a digit;
- `[^ABCD]+` means any string that does not contain *A*, *B*, *C*, or *D*.

Such classes are also called negated character classes.

Range of Characters.

Inside square brackets, we can also specify ranges using the hyphen character: `-`. For example:

- The expression `[1-4]` means any of the digits *1*, *2*, *3*, or *4*, and `a[1-4]b` matches *a1b*, *a2b*, *a3c*, and *a4b*.
- The expression `[a-zàâäæçéèêëîïôöœßùûüÿ]` matches any lowercase accented or unaccented letter of French and German.

Metacharacters.

Inside a character class, the hyphen is a metacharacter describing a range. If we want to search it like an ordinary character and include it in a class, we need to quote it with a backslash like this: `\-`. The expression `[1\-4]` means any of the characters *1*, -, or *4*.

In addition to the hyphen, the other metacharacters used in character classes are: the closing square bracket, `]`, the backslash, `\`, the caret, `^`, and the dollar sign, `$`. As for carets, they need to be quoted to be treated as normal characters in a character class. However, when they are in an unambiguous position, Python will interpret them correctly even without the escape sign. For instance, if the caret is not the first character after the opening bracket, Python will recognize it as a normal character. The expression `[a^b]` matches either *a*, ^, or *b*.

Predefined Character Classes.

Most regex flavors have predefined classes. Table 3.8 lists some useful ones in Python. Some classes are adopted by all the flavors, while some others are specific to Python. In case of doubt, refer to the appropriate documentation. Python also defines classes as properties using the `\p{class}` construct that matches the symbols in `class` and `\P{class}` that matches symbols not in `class`. To name the properties or classes, Python uses its own categories as well as those defined by the Unicode standard that we will review in Chap. 4. This enables the programmer to handle non-Latin scripts more easily.

3.2.6 Nonprintable Symbols or Positions

Some metacharacters match positions and nonprintable symbols. Positions or **anchors** enable one to search a pattern with a specific location in a text. They encode the start and end of a line using, respectively, the caret, `^`, and the dollar symbol, `$`.

The expression `^Chapter` matches lines beginning with *Chapter* and `[0-9]+$` matches lines ending with a number. We can combine both in `^Chapter [0-9]+$`, which matches lines consisting only of the *Chapter* word and a number as *Chapter 3*, for example.

The command line

Table 3.8. Predefined character classes in Perl. After Wall et al. (2000)

Expression	Description	Equivalent	\p{...} equiv.
\d	Any digit	[0-9]	\p{Nd}
\D	Any nondigit	[^0-9]	\P{Nd}
\s	Any whitespace character: space, tabulation, new line, carriage return, or form feed	[\t\n\r\f]	\p{IsSpace}
\S	Any nonwhitespace character	[^\s]	\P{IsSpace}
\w	Any word character: letter, digit, or underscore	[a-zA-Z0-9_]	\p{IsWord}
\W	Any nonword character	[^\w]	\P{IsWord}
\p{IsAlpha}	Any alphabetic character. It includes accented characters		
\p{IsAlnum}	Any alphanumeric character. It includes accented characters	[\p{IsAlpha} \p{Nd}]	
\p{P}	Any punctuation sign		
\p{IsLower}	Any lowercase character. It includes accented characters		
\p{IsUpper}	Any uppercase character. It includes accented characters		

Table 3.9. Some metacharacters matching nonprintable characters in Python

Metachar	Description	Example
^	Matches the start of a line	^ab*c matches ac, abc, abbc, abbbc, etc., when they are located at the beginning of a new line
$	Matches the end of a line	ab?c$ matches ac and abc when they are located at the end of a line
\b	Matches word boundaries	\babc matches abcd but not dabc bcd\b matches abcd but not abcde

```
egrep '^[aeiou]+$' myFile
```

matches the lines of myFile containing only vowels.

Similarly, in Python, the anchor \b matches word boundaries. The expression \bace matches *aces* and *acetylene* but not *place*. Conversely, ace\b matches *place* but neither *aces* nor *acetylene*. The expression \bact\b matches exactly the word *act* and not *react* or *acted*. Table 3.9 summarizes anchors and some nonprintable characters.

From Tables 3.9 and 2.2, you may have noted that the metacharacter \b was used with two different meanings: word boundary or backspace. In fact, its interpretation depends on the context: it is a backspace in character classes; otherwise, it matches word boundaries.

3.2.7 Union and Boolean Operators

We reviewed the basic constructs to write regular expressions. A powerful feature is that we can also combine expressions with operators, as with automata. Using a mathematical term, we say that they define an algebra. Using a simpler analogy, this means that we can arrange regular expressions just like arithmetic expressions. This means, for instance, that it will be possible to apply the repetition metacharacters `*` or `+` not only to the previous character, but to a previous regular expression. This greatly eases the design of complex expressions and makes them very versatile.

Regex languages use three main operators. Two of them are already familiar to us. The first one is the Kleene star or closure, denoted `*`. The second one is the concatenation, which is usually not represented. It is implicit in strings like `abc`, which is the concatenation of characters *a*, *b*, and *c*. To concatenate the word *computer*, a space symbol, and *science*, we just write them in a row: `computer science`.

The third operation is the union and is denoted "`|`". The expression `a|b` means either *a* or *b*. We saw that the regular expression `[Cc]omputer [Ss]cience` could match four strings. We can rewrite an equivalent expression using the union operator: `Computer Science|Computer science|computer Science|computer science`. A union is also called an alternation because the corresponding expression can match any of the alternatives, here four.

3.2.8 Operator Combination and Precedence

Regular expressions and operators are grouped using parentheses. If we omit them, expressions are governed by rules of precedence and associativity. The expression `a|bc` matches the strings *a* and *bc* because the concatenation operator takes precedence over the union. In other words, the concatenation binds the characters stronger than the union. If we want an expression that matches the strings *ac* and *bc*, we need parentheses `(a|b)c`.

Let us examine another example of precedence. We rewrote the expression `[Cc]omputer [Ss]cience` using a union of four strings. Since the difference between expressions lies in the first letters only, we can try to revise this union into something more compact. The character class `[Cc]` is equivalent to the alternation `C|c`, which matches either *C* or *c*. A tentative expression could then be `C|computer S|science`. But it would not match the desired strings; it would find occurrences of either *C*, *computer S*, or *science* because of the operator precedence. We need parentheses to group the alternations `(C|c)omputer (S|s)cience` and thus match the four intended strings.

The order of precedence of the three main operators union, concatenation, and closure is as follows:

1. Closure and other repetition operator (highest);
2. Concatenation, line and word boundaries;
3. Union (lowest).

This entails that `abc*` describes the set *ab*, *abc*, *abcc*, *abccc*, etc. To repeat the pattern *abc*, we need parentheses; and the expression `(abc)*` corresponds to *abc*, *abcabc*, *abcabcabc*, etc.

3.3 Programming with Regular Expressions

We saw that regular expressions were devices to define and search patterns in texts. If we want to use them for more elaborate text processing such as translating characters, substituting words, or counting them, we need a full-fledged programming language, such as Perl, Python, C#, or Java with its `java.util.regex` package. They enable the design of powerful regexes and at the same time, they are complete programming languages.

This section, as well as the next chapter, discusses features of Python. For a review of this language, see Chap. 2.

The two main regex operations are match and substitute. They are often abridged using the Perl regex notations where:

- The `m/regex/` construct denotes a match operation with the regular expression `regex`.
- The `s/regex/replacement/` construct is a substitution operation. This statement matches the first occurrence of `regex` and replaces it by the `replacement` string. If we want to replace all the occurrences of a pattern, we use the `g` modifier, where `g` stands for globally: `s/regex/replacement/g`.

3.3.1 Matching

Python has two regex engines provided by the `re` and `regex` modules. The first one is the standard engine, while the second has extended Unicode capabilities. Outside Unicode, they have similar properties and are roughly interchangeable.

The matching operation, `m/regex/`, is carried out using the `re.search()` function with two arguments: The pattern to search, `regex`, and a string. It returns the first matched object in the string or `None`, if there is no match.

The next program applies `m/ab*c/` to the string *The aerial acceleration alerted the ace pilot*:

```
import regex as re

line = 'The aerial acceleration alerted the ace pilot'
match = re.search('ab*c', line)
match     # <regex.Match object; span=(11, 13), match='ac'>
```

and finds a match object spanning from index 11 to 13 with the value `ac`.

We use the `group()` method to access the matched pattern:

```
match.group() # ac
```

The `re.search()` function stops at the first match. If we want to find all the matches and return them as a list of strings, we use `findall()` instead:

```
match_list = re.findall('ab*c', line)   # ['ac', 'ac']
```

or `finditer()` to return all the match objects:

```
match_iter = re.finditer('ab*c', line)
list(match_iter)
        # [<regex.Match object; span=(11, 13), match='ac'>,
        #  <regex.Match object; span=(36, 38), match='ac'>]
```

In Sect. 3.2, we used the `grep` command to read files, search an expression, and print the lines where we found it. We can use `re.search()` to emulate this command, here with the `m/ab*c/` operation:

```
import sys

for line in sys.stdin:
    if re.search('ab*c', line):    # m/ab*c/
        print('-> ' + line, end='')
```

The program reads from the standard input, `sys.stdin`, and assigns the current line from the input to the `line` variable. The `for` statement reads all the lines until it encounters an end of file. `re.search()` searches the pattern in `line` and returns the first matched object, or `None` if there is no match. The `if` statement tells the program to print the input when it contains the pattern. We run the program to search the file `file_name` with the command:

```
python regex01.py <file_name
```

3.3.2 Match Modifiers

The `re.search()` function supports a set of flags as third argument that modifies the match operation. These flags are equivalent to Perl's `m/regex/modifiers`.

Useful modifiers are:

- Case insensitive: `i`. The instruction `m/regex/i` searches `regex` in the target string regardless of its case. In Python, this corresponds to the flag: `re.I`.
- Multiple lines: `m` (`re.M` in Python). By default, the anchors `^` and `$` match the start and the end of the input string. The instruction `m/regex/m` considers the input string as multiple lines separated by new line characters, where the anchors `^` and `$` match the start and the end of any line in the string.
- Single line: `s` (`re.S` in Python). Normally, a dot symbol "`.`" does not match new line characters. The `/s` modifier makes a dot in the instruction `m/regex/s` match any character, including new lines.

Modifiers can be grouped in any order as in `m/regex/im`, for instance, or `m/regex/sm`, where a dot in `regex` matches any character and the anchors `^` and `$` match just after and before new line characters.

In Python, the modifiers (called flags) are supplied as a sequence separated by vertical bars: `|`.

The next program applies the operation `m/^ab*c/im` to a text: It prints the pattern `ab*c` when it starts a line.

```
text = sys.stdin.read()
match = re.search('^ab*c', text, re.I | re.M) # m/^ab*c/im
if match:
    print('-> ' + match.group())
```

`re.findall()` that finds all the occurrences of a pattern has the same flags as `re.search()`. It is equivalent to the `m/regex/g` (globally) modifier in the PCRE framework.

3.3.3 Substitutions

Python uses the `re.sub()` function to substitute patterns. It has three arguments: `regex`, `replacement`, and `line`, where the substitution occurs. It returns a new line, where by default, it substitutes all the `regex` matches in `line` with `replacement`. Additionally, a fourth parameter, `count`, gives the maximal number of substitutions and a fifth, `flags`, the match modifiers.

We shall write a program to replace all the occurrences of `ab+c` with `ABC` in a file (`s/ab+c/ABC/g`) and print them. We read all the lines of the input; we first check whether they match the regular expression `ab+c`; we then print the old line, substitute the matched pattern, and print the new line:

```
for line in sys.stdin:
    if re.search('ab+c', line):
        print("Old: " + line, end='')
        # Replaces all the occurrences
        line = re.sub('ab+c', 'ABC', line)   # s/ab+c/ABC/g
        print("New: " + line, end='')
```

If we just want to replace the first occurrence, we use this statement instead:

```
        # Replaces the first occurrence
        line = re.sub('ab+c', 'ABC', line, 1) # s/ab+c/ABC/
```

3.3.4 Translating Characters

The transliteration instruction `tr/search_list/replacement_list/`, available in Perl and the Unix shells, replaces all the occurrences of the characters in `search_list` by the corresponding character in `replacement_list`. The instruction `tr/ABC/abc/` replaces the occurrences of *A*, *B*, and *C* by *a*, *b*, and *c*, respectively. The transliteration applied to the string

```
AbCdEfGhIjKlMnOpQrStUvWxYzÉö
```

results in

```
abcdEfGhIjKlMnOpQrStUvWxYzÉö
```

The hyphen specifies a character range, as in the instruction `tr/A-Z/a-z/`, which converts the uppercase characters to their lowercase equivalents.

The transliteration operator, as we defined it, is not a standard component of Python. We can use the similar `str.translate(dict)` function instead, where `dict` is a dictionary that contains the translation mappings. As keys, the dictionary must use the Unicode values of the characters (see Chap. 4). We obtain these values with the `ord()` function. The dictionary implementing `tr/ABC/abc/` is then:

```
dict = {ord('A'):'a', ord('B'):'b', ord('C'):'c'}
```

The string translation is straightforward:

```
line = 'AbCdEfGhIjKlMnOpQrStUvWxYzÉö'
line.translate(dict)     # 'abcdEfGhIjKlMnOpQrStUvWxYzÉö'
```

Encoding the dictionary can be tedious and Python provides a `str.maketrans()` function that creates it automatically with arguments identical to `tr///` as with:

```
dict = str.maketrans('ABC', 'abc')
```

although, it does not interpret the hyphens as ranges.

The instruction `tr` has a few modifiers:

- `d` deletes any characters of the search list that are not found in the replacement list.
- `c` translates characters that belong to the complement of the search list.
- `s` reduces – squeezes, squashes – sequences of characters translated to an identical character to a single instance.

The instruction

```
tr/AEIOUaeiou//d
```

deletes all the vowels and

```
tr/AEIOUaeiou/$/cs
```

replaces all nonvowel characters by a `$` sign. The contiguous sequences of translated dollar signs are reduced to a single sign.

In Python, we can implement the `d` modifier with `str.maketrans()` and its third argument, whose characters are mapped to `None`:

```
dict = str.maketrans('', '', 'AEIOUaeiou')
line.translate(dict)              # 'bCdfGhjKlMnpQrStvWxYzÉö'
```

or more simply with `re.sub()`:

```
re.sub('[AEIOUaeiou]', '', line) # 'bCdfGhjKlMnpQrStvWxYzÉö'
```

The `c` modifier is also easy to implement with `re.sub()` and complemented character classes:

```
re.sub('[^AEIOUaeiou]', '$', line)
         # 'A$$$E$$$I$$$$$O$$$$$U$$$$$$$'
```

and finally, we can emulate the `cs` modifiers with a repetition quantifier:

```
re.sub('[^AEIOUaeiou]{2,}', '$', line)
         # 'A$E$I$O$U$'
```

3.3.5 Back References

It is sometimes useful to keep a reference to matched patterns or parts of them. Let us imagine that we want to find a sequence of three identical characters, which corresponds to matching a character and checking if the next two characters are identical to the first character. To do this, we first tell Python to remember the matched pattern and we put parentheses around it. This creates a buffer to hold the pattern and we refer back to it by the sequence `\1`.

The instruction `m/(.)\1\1/` matches sequences of three identical characters. We obtain the value of the buffer from the returned match object using its `group(1)` method:

```
line = 'abbbcdeeef'
match = re.search(r'(.)\1\1', line)
match.group(1)                  # 'b'
```

We need to use a raw string and the `r` prefix to encode the regex in `search()`, otherwise `\1` would be interpreted as an octal number; see Sect. 2.4.4.

We can use the back-references in substitutions. The instruction

```
s/(.)\1\1/***/g
```

applied to a string matches sequences of three identical characters and replaces them with three stars. In Python:

```
re.sub(r'(.)\1\1', '***', 'abbbcdeeef')  # 'a***cd***f'
```

Python can create as many buffers as we need. It allocates a new one when it encounters a left parenthesis and refers back to it by references `\1`, `\2`, `\3`, etc. The first pair of parentheses corresponds to `\1`, the second pair to `\2`, the third to `\3`, etc. Outside the regular expression, the `\<digit>` reference is returned by `group(<digit>)`: `match_object.group(1)`, `match_object.group(2)`, `match_object.group(3)`, etc.

As an example, let us write a program to extract monetary expressions in phrases like this one:

We'll buy it for $72.40

The regex below matches amounts of money starting with the dollar sign with parentheses around the integer and decimal parts:

```
m/\$ *([0-9]+)\.?([0-9]*)/
```

We extract these two parts in Python with `group()`:

```
price = "We'll buy it for $72.40"
match = re.search('\$ *([0-9]+)\.?([0-9]*)', price)
match.group()           # '$72.40' The entire match
match.group(1)          # '72' The first group
match.group(2)          # '40' The second group
```

We can use these back references directly in a substitution. The next instruction matches the decimal amounts of money expressed with the dollar sign and substitutes them with the words *dollars* and *cents* in clear in the replacement string:

```
s/\$ *([0-9]+)\.?([0-9]*)/\1 dollars and \2 cents/g
```

and in Python:

```
price = "We'll buy it for $72.40"
re.sub('\$ *([0-9]+)\.?([0-9]*)',
       r'\1 dollars and \2 cents', price)
   # We'll buy it for 72 dollars and 40 cents
```

3.3.6 Match Objects

We saw in Sect. 3.3.1 that the `search()` operations result in match objects. We used their `group()` method to return the matched groups, where:

- `match_object.group()` or `match_object.group(0)` return the entire match;
- `match_object.group(n)` returns the nth parenthetized subgroup.

In addition, the `match_object.groups()` returns a tuple with all the groups and the `match_object.string` instance variable contains the input string.

```
price = "We'll buy it for $72.40"
match = re.search('\$ *([0-9]+)\.?([0-9]*)', price)
match.string            # We'll buy it for $72.40
match.groups()          # ('72', '40')
```

We extract the positions of the matched substrings with the functions:

```
match_object.start([group])
match_object.end([group])
```

where `[group]` is the group number and where 0 or no argument means the whole matched substring. We can use them with slices to extract the strings before and after a matched pattern as in this program:

```
line = """Tell me, O muse, of that ingenious hero
  who travelled far and wide after he had sacked
  the famous town of Troy."""

match = re.search(',.*,', line, re.S)
line[0:match.start()]             # 'Tell me'
line[match.start():match.end()]   # ', O muse,'
line[match.end():]   # 'of that ingenious hero
         #  who travelled far and wide after he had sacked
         #  the famous town of Troy.'
```

3.3.7 Regular Expressions and Strings

Regular expressions and strings are very similar constructs in Python. We already examined the syntax of regex literals and their metacharacters in Sect. 3.2. In addition, as in strings, regexes can use the escape sequences defined in Table 2.2 to match nonprintable or numerically-encoded characters.

If we want to create regular expressions with variables inside like in:

```
'.{0,width}pattern.{0,width}'
```

where `pattern` and `width` are variables, we can use the `str.format()` function (see Sect. 2.4.5).

For example, Python replaces the variables

```
pattern = 'my string'
width = 20
```

with their values in:

```
'.{{0,{1}}}{0}.{{0,{1}}}'.format(pattern, width)
```

to produce

```
'.{0,20}my string.{0,20}'
```

that matches the pattern *my string* with 0 to 20 characters to the left and to the right. Note that we escaped the literal curly braces by doubling them.

Instead of using indices, we can also name the variables as in:

```
('.{{0,{width}}}{pattern}.{{0,{width}}}'
 .format(pattern=pattern, width=width))
```

3.4 Finding Concordances

Concordances of a word, an expression, or more generally any string in a corpus are easy to obtain with Python. In our programs, we will represent the corpus as

one single big string, and concordancing will simply consist in matching the pattern we are searching as a substring of the whole list. There will be no need then to consider the corpus structure, that is, whether it is made of blanks, words, sentences, or paragraphs.

3.4.1 Concordances in Python

To have a convenient input of the concordance parameters – the file name, the pattern to search, and the span size of the concordance – we will design the Python program so that it can read them from the command line as in

```
python concordance.py corpus.txt pattern_to_search 15
```

These arguments are passed to Python by the operating system in the form of a list.

Now let us write a concordance program inspired by Cooper (1999). We use three arguments in the command line: the file name, the pattern to search, and the span size. Python reads them and stores them in an list with the reserved name: `sys.argv[1:]`. We assign these arguments, respectively, to `file_name`, `pattern`, and `width`.

We open the file using the `open()` function, read all the text and we assign it to the `text` variable. If `open()` fails, the program exits using `except` and prints a message to inform us that it could not open the file.

In addition to single words, we may want to search concordances of a phrase such as *the Achaeans*. Depending on the text formatting, the phrase's words can be on the same line or spread on two lines of text as in:

```
I see that the Achaeans are subject to you in great
multitudes.
...
the banks of the river Sangarius; I was their ally,
and with them when the Amazons, peers of men, came up
against them, but even they were not so many as the
Achaeans."
```

The Python string `'the Achaeans'` matches the first occurrence of the phrase in the text, but not the second one as the two words are separated by a line break.

There are two ways to cope with that:

1. We can modify `pattern`, the phrase to search, so that it matches across sequences of line breaks, tabulations, or spaces. To do this, we replace the sequences of spaces in `pattern` with the generic white space character class: `s/ +/\\s+/g`.
2. The second possibility is to normalize the text, `text`, so that the line breaks and all kinds white spaces in the text are replaced with a standard space: `s/\s+/ /g`.

Both solutions can deal with the multiple conventions to mark line breaks, the two most common ones being `\n` and `\r\n` adopted, respectively, by Unix and Windows. Moreover, the text normalization makes it easier to format the concordance output and print the results. In our program, we will keep both instructions, although they are somewhat redundant.

Finally, we write a regular expression that adds `width` characters to the left and to the right of the searched pattern. We create a back reference by setting parentheses around the whole expression and we print its value stored in `group(1)`. Applying the `re.search()` method would stop at the first occurrence of the concordance. To find them all in `text`, we use the `re.finditer()` method that returns all the match objects in the form of an iterator. We use a `for` loop to print them.

```
import re
import sys

[file_name, pattern, width] = sys.argv[1:]
try:
    text = open(file_name).read()
except:
    print('Could not open file', file_name)
    exit(0)

# spaces match tabs and newlines
pattern = re.sub(' ', '\\s+', pattern)
# line breaks and blank sequences are replaced by spaces
text = re.sub('\s+', ' ', text)
concordance = ('(.{{0,{width}}}{pattern}.{{0,{width}}})'
               .format(pattern=pattern, width=width))
for match in re.finditer(concordance, text):
    print(match.group(1))
```

Now let us run the command:

```
python concordance.py odyssey.txt Penelope 25

he suitors of his mother Penelope, who persist in eating u
ace dying out yet, while Penelope has such a fine son as y
 laid upon the Achaeans. Penelope, daughter of Icarius, he
 blood of Ulysses and of Penelope in your veins I see no l
 his long-suffering wife Penelope, and his son Telemachus,
ngs. It was not long ere Penelope came to know what the su
he threshold of her room Penelope said: "Medon, what have
```

3.4.2 Lookahead

The `finditer()` function, just like `findall()`, scans the text from left to right and finds all the nonoverlapping matches. When the regular expression matches a

concordance in the text, the search is started again from the end index of the match (see Sect. 3.3.6).

This nonoverlapping match means that when the interval between two patterns is smaller than twice the width of the left or right context, `width`, the concordance is truncated. Searching the two occurrences of *Achaeans*, which are 31 characters apart, in this text from the *Iliad*:

> You forget this, and threaten to rob me of the prize for which I have toiled, and which the sons of the *Achaeans* have given me. Never when the *Achaeans* sack any rich city of the Trojans do I receive so good a prize as you do, though it is my hands that do the better part of the fighting.

with a width of 25 characters shows an example of this. Applying

```
python concordance.py iliad.txt Achaeans 25
```

produces:

```
nd which the sons of the Achaeans have given me. Never whe
n the Achaeans sack any rich city of th
```

where the left context is incomplete.

To show the complete concordance, we need a special kind of match called **lookahead** that tells the regex to match `width` characters to the right, but not to advance the end index. This lookahead match is denoted by the `(?=...)` construct.

The concordance regex we need is then:

```
'(.{0,25}Achaeans(?=.{0,25}))'
```

However, the backreference does not capture the lookahead and we need to create a specific one by surrounding it with parentheses inside the construct:

```
'(.{0,25}Achaeans(?=(.{0,25})))'
```

The left part of the concordance is then stored in `group(1)` and the right one, the lookahead, in `group(2)`.

If we want to interpolate the `pattern` and `width` variables, the complete regex is:

```
'(.{{0,{width}}}{pattern}(?=(.{{0,{width}}})))'.
 format(pattern=pattern, width=width))
```

3.5 Approximate String Matching

So far, we have used regular expressions to match exact patterns. However, in many applications, such as in spell checkers, we need to extend the match span to search a set of related patterns or strings. In this section, we review techniques to carry out approximate or inexact string matching.

Table 3.10. Typographical errors (typos) and corrections. Strings differ by one operation. The *correction* is the source and the *typo* is the target. Unless specified, other operations are just copies. After Kernighan et al. (1990)

Typo	Correction	Source	Target	Position	Operation
acress	actress	–	t	2	Deletion
acress	cress	a	–	0	Insertion
acress	caress	ac	ca	0	Transposition
acress	access	r	c	2	Substitution
acress	across	e	o	3	Substitution
acress	acres	s	–	4	Insertion
acress	acres	s	–	5	Insertion

3.5.1 Edit Operations

A common method to create a set of related strings is to apply a sequence of edit operations that transforms a source string s into a target string t. The operations are carried out from left to right using two pointers that mark the position of the next character to edit in both strings:

- The copy operation is the simplest. It copies the current character of the source string to the target string. Evidently, the repetition of copy operations produces equal source and target strings.
- Substitution replaces one character from the source string by a new character in the target string. The pointers are incremented by one in both the source and target strings.
- Insertion inserts a new character in the target string. The pointer in the target string is incremented by one, but the pointer in the source string is not.
- Deletion deletes the current character in the target string, i.e., the current character is not copied in the target string. The pointer in the source string is incremented by one, but the pointer in the target string is not.
- Reversal (or transposition) copies two adjacent characters of the source string and transposes them in the target string. The pointers are incremented by two characters.

Kernighan et al. (1990) illustrate these operations with the misspelled word *acress* and its possible corrections (Table 3.10). They named the transformations from the point of view of the correction, not from the typo.

3.5.2 Edit Operations for Spell Checking

Spell checkers identify the misspelled or unknown words in text. They are ubiquitous tools that we now find in almost all word processors, messaging applications, editors, etc. Spell checkers start from a pre-defined vocabulary (or dictionary) of correct words, scan the words from left to right, look them up in their dictionary, and for the words outside the vocabulary, supposedly typos, suggest corrections.

Given a typo, spell checkers find words from their vocabulary that are close in terms of edit distance. To carry this out, they typically apply edit operations to the typo to generate a set of new strings called "edits." They then look up these edits in the dictionary, discard the unknown ones, and propose the rest to the user as possible corrections.

If we allow only one edit operation on a source string of length n, and if we consider an alphabet of 26 unaccented letters, the deletion will generate n new strings; the insertion, $(n+1) \times 26$ strings; the substitution, $n \times 25$; and the transposition, $n-1$ new strings. In the next sections, we examine how to generate these candidates in Python.

Generating correction candidates is easy in Python. We propose here an implementation by Norvig (2007)[1] that uses list comprehensions: One comprehension per edit operation: delete, transpose, replace, and insert. The `edit()` function first splits the input, the unknown word, at all possible points and then applies the operations to the list of splits. See Sect. 2.9 for a description. Finally, the `set()` function returns a list of unique candidates.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
    splits = [(word[:i], word[i:])
              for i in range(len(word) + 1)]
    deletes = [a + b[1:] for a, b in splits if b]
    transposes = [a + b[1] + b[0] + b[2:]
                  for a, b in splits if len(b) > 1]
    replaces = [a + c + b[1:]
                for a, b in splits for c in alphabet if b]
    inserts = [a + c + b for a, b in splits for c in alphabet]
    return set(deletes + transposes + replaces + inserts)
```

Applying `edits1()` to *acress* returns a list of 336 unique candidates that includes the dictionary words in Table 3.10 and more than 330 other strings such as: *aeress*, *hacress*, *acrecs*, *acrehss*, *acwress*, *acrses*, etc.

Now how do we extract acceptable words from the full set of edits? In his program, Norvig (2007) uses a corpus of one million words to build a vocabulary (see Sect. 6.4.3 for a program to carry this out); the candidates are looked up in this dictionary to find the possible corrections. When there is more than one valid candidate, Norvig (2007) ranks them by the frequencies he observed in the corpus.

For *acress*, the edit operations yield five possible corrections listed below, where the figure is the word frequency in the corpus:

```
{'caress':4, 'across':223, 'access':57, 'acres':37, 'actress':8}
```

The spell checker proposes *across* as correction as it is the most frequent word. We also note that *cress* is not in the list as this word was is not in the corpus.

[1] Under MIT license, `http://www.opensource.org/licenses/mit-license.php`.

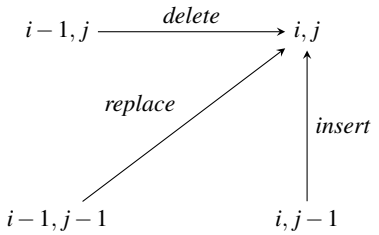

Fig. 3.1. Edit operations

If `edits1()` does not generate any known word, we can reapply it to the list of edits. Studies showed that most typos can be corrected with less than two edit operations (Norvig, 2007).

3.5.3 Minimum Edit Distance

Complementary to edit operations, edit distances measure the similarity between strings. They assign a cost to each edit operation, usually 0 to copies and 1 to deletions and insertions. Substitutions and transpositions correspond both to an insertion and a deletion. We can derive from this that they each have a cost of 2. Edit distances tell how far a source string is from a target string: the lower the distance, the closer the strings.

Given a set of edit operations, the minimum edit distance is the operation sequence that has the minimal cost needed to transform the source string into the target string. If we restrict the operations to copy/substitute, insert, and delete, we can represent the edit operations using a table, where the distance at a certain position in the table is derived from distances in adjacent positions already computed. This is expressed by the formula:

$$edit\_distance(i,j) = \min \begin{pmatrix} edit\_distance(i-1,j) + del\_cost \\ edit\_distance(i-1,j-1) + subst\_cost \\ edit\_distance(i,j-1) + ins\_cost \end{pmatrix}.$$

The boundary conditions for the first row and the first column correspond to a sequence of deletions and of insertions. They are defined as $edit\_distance(i,0) = i$ and $edit\_distance(0,j) = j$.

We compute the cell values as a walk through the table from the beginning of the strings at the bottom left corner, and we proceed upward and rightward to fill adjacent cells from those where the value is already known. Arrows in Fig. 3.5.3 represent the three edit operations, and Table 3.11 shows the distances to transform *language* into *lineage*. The value of the minimum edit distance is 5 and is shown at the upper right corner of the table.

The minimum edit distance algorithm is part of the **dynamic programming** techniques. Their principles are relatively simple. They use a table to represent data, and

Table 3.11. Distances between *language* and *lineage*

e	7	6	5	6	5	6	7	6	**5**
g	6	5	4	5	4	5	6	5	6
a	5	4	3	4	5	6	5	6	7
e	4	3	4	3	4	5	6	7	6
n	3	2	3	2	3	4	5	6	7
i	2	1	2	3	4	5	6	7	8
l	1	0	1	2	3	4	5	6	7
Start	0	1	2	3	4	5	6	7	8
–	**Start**	**l**	**a**	**n**	**g**	**u**	**a**	**g**	**e**

they solve a problem at a certain point by combining solutions to subproblems. Dynamic programming is a generic term that covers a set of widely used methods in optimization.

3.5.4 Computing the Minimum Edit Distance in Python

To implement the minimum edit distance in Python, we use the `len()` built-in function to compute the length of the source and target.

```
import sys

[source, target] = sys.argv[1:]

length_s = len(source) + 1
length_t = len(target) + 1

# Initialize first row and column
table = [None] * length_s

for i in range(length_s):
    table[i] = [None] * length_t
    table[i][0] = i
for j in range(length_t):
    table[0][j] = j

# Fills the table. Start index of rows and columns is 1
for i in range(1, length_s):
    for j in range(1, length_t):
        # Is it a copy or a substitution?
        cost = 0 if source[i - 1] == target[j - 1] else 2
        # Computes the minimum
        minimum = table[i - 1][j - 1] + cost
        if minimum > table[i][j - 1] + 1:
```

	First alignment	Third alignment
Without epsilon symbols	l a n g *u* a g e | | | | / / / l i n e a g e	l a n *g* *u* a g e | | | / / / l i n *e* a g e
With epsilon symbols	l a n g u a g e | | | | | | | | l i n e *ε* a g e	l a n g u *ε* a g e | | | | | | | | | l i n *ε* *ε* e a g e

Fig. 3.2. Alignments of *lineage* and *language*. The figure contains two possible representations of them. In the *upper row*, the deletions in the source string are in italics, as are the insertions in the target string. The *lower row* shows a synchronized alignment, where deletions in the source string as well as the insertions in the target string are aligned with epsilon symbols (null symbols)

```
            minimum = table[i][j - 1] + 1
        if minimum > table[i - 1][j] + 1:
            minimum = table[i - 1][j] + 1
        table[i][j] = minimum

print('Minimum distance: ', table[length_s - 1][length_t - 1])
```

3.5.5 Searching Edits

Once we have filled the table, we can search the operation sequences that correspond to the minimum edit distance. Such a sequence is also called an **alignment**. Table 3.2 shows two examples of them.

A frequently used technique is to consider each cell in Table 3.11 and to store the coordinates of all the adjacent cells that enabled us to fill it. For instance, the program filled the last cell of coordinates $(8,7)$, containing 5 (`table[8][7]`), using the content of cell $(7,6)$. The storage can be a parallel table, where each cell contains the coordinates of the immediately preceding positions (the backpointers). Starting from the last cell down to the bottom left cell, $(0,0)$, we traverse the table from adjacent cell to adjacent cell to recover all the alignments. This program is left as an exercise (Exercise 3.7).

3.6 Further Reading

Corpora are now easy to obtain. Organizations such as the Linguistic Data Consortium and ELRA collect and distribute texts in many languages. Although not widely cited, Busa (1974, 1996) is the author of the first large computerized corpus, the *Index Thomisticus*, a complete edition of the works of Saint Thomas Aquinas. The corpus, which is entirely lemmatized, is available online (`http:`

//www.corpusthomisticum.org/). FranText is also a notable early corpus of more than 100 million words. It helped write the *Trésor de la langue française* (Imbs and Quemada, 1971–1994), a comprehensive French dictionary. Other early corpora include the Bank of English, which contributed to the *Collins COBUILD Dictionary* (Sinclair, 1987).

Concordancing plays a role today that goes well beyond lexicography. Google, Bing, and other web search engines can be considered as modern avatars of concordancers as they return a small passage – a snippet – of a document, where a phrase or words are cited. The Dominicans who created the first concordances in the thirteenth century surely did not forecast the future of their brainchild and the billions of searches per day it would entail. For a history of early concordances to the scriptures, see Rouse and Rouse (1974).

The code examples in these chapter enabled us to search strings and patterns in a corpus. For large volumes of text, a more realistic application would first index all the words before a user can search them. Manning et al. (2008) is a good review of indexing techniques. Lucene (`http://lucene.apache.org/`) is a widely used system to carry out text indexing and search.

Text and corpus analysis are an active focus of research in computational linguistics. Kaeding (1897) and Estoup (1912), the latter cited in Petruszewycz (1973), were among the pioneers in this field, at the turn of the 20th century, when they used corpora to carry out systematic studies on letter and word frequencies for stenography. Paradoxically, natural language processing conducted by computer scientists largely ignored corpora until the 1990s, when they rediscovered techniques routinely used in the humanities. For a short history, see Zampolli (2003) and Busa (2009).

Roche and Schabes (1997, Chap. 1) is a concise and clear introduction to automata theory. It makes extensive use of mathematical notations, however. Hopcroft et al. (2007) is a standard and comprehensive textbook on automata and regular expressions. Friedl (2006) is a thorough presentation of regular expressions oriented toward programming techniques and applications. Goyvaerts and Levithan (2012) is another good book on the same topic that comes with a very complete web site: `https://www.regular-expressions.info/`. Finally, the `https://regex101.com/` site provides an excellent tool to experiment visually with regular expressions.

Although the idea of automata underlies some mathematical theories of the 19th century (such as those of Markov, Gödel, or Turing), Kleene (1956) was the first to give a formal definition. He also proved the equivalence between regular expressions and FSA. Thompson (1968) was the first to implement a widely used editor embedding a regular expression tool: Global/Regular Expression/Print, better known as `grep`.

There are several FSA toolkits available from the Internet. The Perl Compatible Regular Expressions (PCRE) library is an open-source set of functions that implements the Perl regex syntax. It is written in C by Philip Hazel (`http://www.pcre.org/`). The FSA utilities (van Noord and Gerdemann, 2001) is a Prolog package to manipulate regular expressions, automata, and transducers (`http://odur.let.rug.nl/~vannoord/Fsa/`). The OpenFst library (Mohri et al., 2000; Allauzen et al., 2007) is another set of tools (`http://www.openfst.org/`). Both

include rational operations – union, concatenation, closure, reversal – and equivalence transformation – ε-elimination, determinization, and minimization.

Exercises

3.1. Write a regular expression that finds occurrences of *honour* and *honor* in a text.

3.2. Write a regular expression that finds lines containing all the vowels *a*, *e*, *i*, *o*, *u*, in that order.

3.3. Write a regular expression that finds lines consisting only of letters *a*, *b*, or *c*.

3.4. List the strings generated by the expressions:

```
(ab)*c
(a.)*c
(a|b)*
a|b*|(a|b)*a
a|bc*d
```

3.5. Complement the Python concordance program to sort the lines according to words appearing on the right of the string to search.

3.6. Write the iterative deepening search in Python to find the minimum edit distance.

3.7. Extend the Python program in Sect. 3.5.4 to find the alignments. See the last paragraph of Sect. 3.5.5 for an idea of the algorithm.

4

Encoding and Annotation Schemes

ʽΕλλάδι φωνήεντα καὶ ἔμφρονα δῶρα κομίζων
γλώσσης ὄργανα τεῦξεν ὁμόθροα, συμφυέος δὲ
ἁρμονίης στοιχηδὸν ἐς ἄζυγα σύζυγα μίξας
γραπτὸν ἀσιγήτοιο τύπον τορνώσατο σιγῆς,
πάτρια θεσπεσίης δεδαημένος ὄργια τέχνης,

Nonnus Panopolitanus, *Dionysiaca*, Book IV, verses 261–265. Fifth century.

But Cadmos [from Sidon in Phoenicia] brought gifts of voice and thought for all Hellas; he fashioned tools to echo the sounds of the tongue, he mingled sonant and consonant in one order of connected harmony. So he rounded off a graven model of speaking silence; for he had learnt the secrets of his country's sublime art.

Translation W. H. D. Rouse. Loeb Classical Library.

4.1 Encoding Texts

At the most basic level, computers only understand binary digits and numbers. Corpora as well as any computerized texts have to be converted into a digital format to be read by machines. From their American early history, computers inherited encoding formats designed for the English language. The most famous one is the **American Standard Code for Information Interchange** (ASCII). Although well established for English, the adaptation of ASCII to other languages led to clunky evolutions and many variants. It ended (temporarily?) with Unicode, a universal scheme compatible with ASCII and intended to cover all the scripts of the world.

We saw in Chap. 3 that some corpora include linguistic information to complement raw texts. This information is conveyed through annotations that describe quantities of structures. They range from text organization, such as titles, paragraphs, and sentences, to semantic information including grammatical data, part-of-speech

Table 4.1. The ASCII character set arranged in a table consisting of 6 rows and 16 columns. We obtain the ASCII code of a character by adding the first number of its row and the number of the column. For instance, *A* has the decimal code 64 + 1 = 65, and *e* has the code 96 + 5 = 101

	0	1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

labels, or syntactic structures, etc. In contrast to character encoding, no annotation scheme has yet reached a level where it can claim to be a standard. However, the **Extensible Markup Language** (XML), a language to define annotations, is well underway to unify them under a shared markup syntax. XML in itself is not an annotation language. It is a scheme that enables users to define annotations within a specific framework.

In this chapter, we will introduce the most useful character encoding schemes and review the basics of XML. We will examine related topics of standardized presentation of time and date, and how to sort words in different languages. We will then outline two significant theoretical concepts behind codes – entropy and perplexity – and how they can help design efficient codes. Entropy is a very versatile measure with many applications. We will use it to learn automatically decision trees from data and build classifiers. This will enable us to review our first machine-learning algorithm. Machine learning is now instrumental in most areas of natural language processing and we will conclude this chapter with the description of three other linear classifiers that are among the most widely used.

4.2 Character Sets

4.2.1 Representing Characters

Words, at least in European languages, consist of characters. Prior to any further digital processing, it is necessary to build an encoding scheme that maps the character or symbol repertoire of a language to numeric values – integers. The Baudot code is one of the oldest electric codes. It uses five bits and hence has the capacity to represent $2^5 = 32$ characters: the Latin alphabet and some control commands like the carriage return and the bell. The ASCII code uses seven bits. It can represent $2^7 = 128$ symbols with positive integer values ranging from 0 to 127. The characters use the contiguous positions from 32 to 126. The values in the range [0..31] and 127 correspond to controls used, for instance, in data transmission (Table 4.1).

Table 4.2. Characters specific to French and German

	French	**German**
Lowercase	à â æ ç é è ê ë î ï ô œ ù û ü ÿ	ä ö ü ß
Uppercase	À Â Æ Ç É È Ê Ë Î Ï Ô Œ Ù Û Ü Ÿ	Ä Ö Ü

Table 4.3. The ISO Latin 1 character set (ISO-8859-1) covering most characters from Western European languages

	0	1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Table 4.4. The ISO Latin 9 character set (ISO-8859-15) that replaces rare symbols from Latin 1 with the characters œ, Œ, š, Š, ž, Ž, Ÿ, and €. The table only shows rows that differ from Latin 1

	0	1	2	4	3	5	6	7	8	9	10	11	12	13	14	15
160		¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	-	®	¯
176	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿

ASCII was created originally for English. It cannot handle other European languages that have accented letters, such as *é*, *à*, or other diacritics like *ø* and *ä*, not to mention languages that do not use the Latin alphabet. Table 4.2 shows characters used in French and German that are ignored by ASCII. Most computers used to represent characters on octets – words of eight bits – and ASCII was extended with the eighth unoccupied bit to the values in the range [128..255] ($2^8 = 256$). Unfortunately, these extensions were not standardized and depended on the operating system. The same character, for instance, *ê*, could have a different encoding in the Windows, Macintosh, and Unix operating systems.

The ISO Latin 1 character set (ISO-8859-1) is a standard that tried to reconcile Western European character encodings (Table 4.3). Unfortunately, Latin 1 was ill-designed and forgot characters such as the French *Œ*, *œ*, the German quote „, or the Dutch *ij*, *IJ*. Operating systems such as Windows and Mac OS used a variation of it that they had to complement with the missing characters. Later, ISO Latin 9 (ISO-8859-15) updated Latin 1 (Table 4.4). It restored forgotten French and Finnish characters and added the euro currency sign, €.

4.2.2 Unicode

While ASCII has been very popular, its 128 positions could not support the characters of many languages in the world. Therefore a group of companies formed a consortium to create a new, universal coding scheme: Unicode. Unicode has quickly replaced older encoding schemes, and Windows, Mac OS, and Java platforms have now adopted it while sometimes ensuring backward compatibility.

The initial goal of Unicode was to define a superset of all other character sets, ASCII, Latin 1, and others, to represent all the languages of the world. The Unicode consortium has produced character tables of most alphabets and scripts of European, Asian, African, and Near Eastern languages, and assigned numeric values to the characters. Unicode started with a 16-bit code that could represent up to 65,000 characters. The code was subsequently extended to 32 bits with values ranging from 0 to 10FFFF in hexadecimal. This Unicode code space has then a capacity of 1,114,112 characters.

The standardized set of Unicode characters is called the universal character set (UCS). It is divided into several planes, where the basic multilingual plane (BMP) contains all the common characters, with the exception of some Chinese ideograms. Characters in the BMP fit on a 2-octet code (UCS-2). The 4-octet code (UCS-4) can represent, as we saw, more than a million characters. It covers all the UCS-2 characters and rare characters: historic scripts, some mathematical symbols, private characters, etc.

Unicode groups characters or symbols by script – Latin, Greek, Cyrillic, Hebrew, Arabic, Indic, Japanese, Chinese – and identifies each character by a single hexadecimal number, called a code point, and a name as

```
U+0041 LATIN CAPITAL LETTER A
U+0042 LATIN CAPITAL LETTER B
U+0043 LATIN CAPITAL LETTER C

...

U+0391 GREEK CAPITAL LETTER ALPHA
U+0392 GREEK CAPITAL LETTER BETA
U+0393 GREEK CAPITAL LETTER GAMMA
```

The `U+` symbol means that the number after it corresponds to a Unicode position.

Unicode allows the composition of accented characters from a base character and one or more diacritics. That is the case for the French $\hat{E}$ or the Scandinavian $\mathring{A}$. Both characters have a single code point:

```
U+00CA LATIN CAPITAL LETTER E WITH CIRCUMFLEX
U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
```

They can also be defined as a sequence of two keys: E + ˆ and A + ˚, corresponding to respectively to

```
U+0045 LATIN CAPITAL LETTER E
U+0302 COMBINING CIRCUMFLEX ACCENT
```

and

```
U+0041 LATIN CAPITAL LETTER A
U+030A COMBINING RING ABOVE
```

The resulting graphical symbol is called a grapheme. A grapheme is a "natural" character or a symbol. It may correspond to a single code point as E or A, or result from a composition as $\hat{E}$ or $\mathring{A}$.

Unicode allocates contiguous blocks of code to scripts from `U+0000`. They start with alphabetic scripts: Latin, Greek, Cyrillic, Hebrew, Arabic, etc., then the symbols area, and Asian ideograms or alphabets. Ideograms used by the Chinese, Japanese, and Korean (CJK) languages are unified to avoid duplication. Table 4.5 shows the script allocation. The space devoted to Asian scripts occupies most of the table.

4.2.3 Unicode Character Properties

Unicode associates a list of properties to each code point. This list is defined in the Unicode character database and includes the name of the code point (character name), its so-called general category – whether it is a letter, digit, punctuation, symbol, mark, or other – the name of its script, for instance Latin or Arabic, and its code block (The Unicode Consortium, 2012).

Each property has a set of possible values. Table 4.6 shows this set for the general category, where each value consists of one or two letters. The first letter is a major class and the second one, a subclass of it. For instance, `L` corresponds to a letter, `Lu` to an uppercase letter; `Ll`, to a lowercase letter, while `N` corresponds to a number and `Nd`, to a number, decimal digit.

We can use these Unicode properties in Python regular expressions to search characters, categories, blocks, and scripts by their names. We need to import the `regex` module however, instead of the standard `re`. We match a specific code point with the `\N{name}` construct, where `name` is the name of the code point, or with its hexadecimal `\uxxxx` code (see Table 2.2) as:

- `\N{LATIN CAPITAL LETTER E WITH CIRCUMFLEX}` and `\u00CA` that match Ê and
- `\N{GREEK CAPITAL LETTER GAMMA}` and `\u0393` that match Γ.

We match code points in blocks, categories, and scripts with the `\p{property}` construct introduced in Sect. 3.2.5, or its complement `\P{property}` to match code points without the property:

For a block, we build a Python regex by replacing `property` with the block name in Table 4.5. Python also requires an `In` prefix and that white spaces are replaced with underscores as `InBasic_Latin` or `InLatin_Extended-A`.
For example, `\p{InGreek_and_Coptic}` matches code points in the Greek and

Table 4.5. Unicode subrange allocation of the universal character set (simplified)

Code	Name	Code	Name
0000	Basic Latin	1400	Unified Canadian Aboriginal Syllabics
0080	Latin-1 Supplement	1680	Ogham
0100	Latin Extended-A	16A0	Runic
0180	Latin Extended-B	1780	Khmer
0250	IPA Extensions	1800	Mongolian
02B0	Spacing Modifier Letters	1E00	Latin Extended Additional
0300	Combining Diacritical Marks	1F00	Greek Extended
0370	Greek and Coptic	2000	General Punctuation
0400	Cyrillic	2800	Braille Patterns
0500	Cyrillic Supplement	2E80	CJK Radicals Supplement
0530	Armenian	2F00	Kangxi Radicals
0590	Hebrew	3000	CJK Symbols and Punctuation
0600	Arabic	3040	Hiragana
0700	Syriac	30A0	Katakana
0750	Arabic Supplement	3100	Bopomofo
0780	Thaana	3130	Hangul Compatibility Jamo
07C0	NKo	3190	Kanbun
0800	Samaritan	31A0	Bopomofo Extended
0900	Devanagari	3200	Enclosed CJK Letters and Months
0980	Bengali	3300	CJK Compatibility
0A00	Gurmukhi	3400	CJK Unified Ideographs Extension A
0A80	Gujarati	4E00	CJK Unified Ideographs
0B00	Oriya	A000	Yi Syllables
0B80	Tamil	A490	Yi Radicals
0C00	Telugu	AC00	Hangul Syllables
0C80	Kannada	D800	High Surrogates
0D00	Malayalam	E000	Private Use Area
0D80	Sinhala	F900	CJK Compatibility Ideographs
0E00	Thai	10000	Linear B Syllabary
0E80	Lao	10140	Ancient Greek Numbers
0F00	Tibetan	10190	Ancient Symbols
1000	Myanmar	10300	Old Italic
10A0	Georgian	10900	Phoenician
1100	Hangul Jamo	10920	Lydian
1200	Ethiopic	12000	Cuneiform
13A0	Cherokee	100000	Supplementary Private Use Area-B

Coptic block whose Unicode range is `[0370..03FF]`. This roughly corresponds to the Greek characters. However, some of the code points in this block are not assigned and some others are Coptic characters.

For a general category, we use either the short or long names in Table 4.6 as `Letter` or `Lu`. For example, `\p{Currency_Symbol}` matches currency symbols and `\P{L}` all nonletters.

Table 4.6. Values of the general category with their short and long names. The *left column* lists to the major classes, and the *right* one the subclasses. After The Unicode Consortium (2012)

Major classes		**Subclasses**	
Short	**Long**	**Short**	**Long**
L	Letter		
		Lu	Uppercase_Letter
		Ll	Lowercase_Letter
		Lt	Titlecase_Letter
		Lm	Modifier_Letter
		Lo	Other_Letter
M	Mark		
		Mn	Nonspacing_Mark
		Mc	Spacing_Mark
		Me	Enclosing_Mark
N	Number		
		Nd	Decimal_Number
		Nl	Letter_Number
		No	Other_Number
P	Punctuation		
		Pc	Connector_Punctuation
		Pd	Dash_Punctuation
		Ps	Open_Punctuation
		Pe	Close_Punctuation
		Pi	Initial_Punctuation
		Pf	Final_Punctuation
		Po	Other_Punctuation
S	Symbol		
		Sm	Math_Symbol
		Sc	Currency_Symbol
		Sk	Modifier_Symbol
		So	Other_Symbol
Z	Separator		
		Zs	Space_Separator
		Zl	Line_Separator
		Zp	Paragraph_Separator
C	Control		
		Cc	Control
		Cf	Format
		Cs	Surrogate
		Co	Private_Use
		Cn	Unassigned

For a script, we use its name in Table 4.7. The regex will match all the code points belonging to this script, even if they are scattered in different blocks. For example, the regex `\p{Greek}` matches the Greek characters in the Greek

and Coptic, Greek Extended, and Ancient Greek Numbers blocks, respectively `[0370..03FF]`, `[1F00..1FFF]`, and `[10140..1018F]`, ignoring the unassigned code points of these blocks and characters that may belong to another script, here Coptic.

Practically, the three instructions below match lines consisting respectively of ASCII characters, of characters in the Greek and Coptic block, and of Greek characters:

```
import regex as re

alphabet = 'αβγδεζηθικλμνξοπρστυφχψω'
match = re.search('^\p{InBasic_Latin}+$', alphabet)
match     # None
match = re.search('^\p{InGreek_and_Coptic}+$', alphabet)
match     # matches alphabet
match = re.search('^\p{Greek}+$', alphabet)
match     # matches alphabet
```

Or a specific Unicode character:

```
match = re.search('\N{GREEK SMALL LETTER ALPHA}', alphabet)
match     # matches 'α'
match = re.search('α', alphabet)
match     # matches 'α'
```

Moreover, following Perl, Python maintains a list of classes that corresponds to synonyms of Unicode properties or to composite properties. Table 3.8 in Chap. 3 showed some of these classes using the `\p{...}` syntax. For example,

- `\p{IsAlpha}` is equivalent to `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}]`;
- `\p{IsAlnum}` is equivalent to `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}]`.

Perl created such composite properties to provide equivalent classes available in other regular expression languages, like the POSIX regular expressions.

4.2.4 The Unicode Encoding Schemes

Unicode offers three major different encoding schemes: UTF-8, UTF-16, and UTF-32. The UTF schemes – Unicode transformation format – encode the same data by units of 8, 16, or 32 bits and can be converted from one to another without loss.

UTF-16 was the original encoding scheme when Unicode started with 16 bits. It uses fixed units of 16 bits – 2 bytes – to encode directly most characters. The code units correspond to the sequence of their code points using precomposed characters, such as *Ê* in *FÊTE*

```
0046 00CA 0054 0045
```

or composing it as with E+ˆ in FEˆTE

Table 4.7. Unicode script names

Arabic Armenian Avestan Balinese Bamum Bengali Bopomofo Braille Buginese Buhid Canadian_Aboriginal Carian Cham Cherokee Common Coptic Cuneiform Cypriot Cyrillic Deseret Devanagari Egyptian_Hieroglyphs Ethiopic Georgian Glagolitic Gothic Greek Gujarati Gurmukhi Han Hangul Hanunoo Hebrew Hiragana Imperial_Aramaic Inherited Inscriptional_Pahlavi Inscriptional_Parthian Javanese Kaithi Kannada Katakana Kayah_Li Kharoshthi Khmer Lao Latin Lepcha Limbu Linear_B Lisu Lycian Lydian Malayalam Meetei_Mayek Mongolian Myanmar New_Tai_Lue Nko Ogham Ol_Chiki Old_Italic Old_Persian Old_South_Arabian Old_Turkic Oriya Osmanya Phags_Pa Phoenician Rejang Runic Samaritan Saurashtra Shavian Sinhala Sundanese Syloti_Nagri Syriac Tagalog Tagbanwa Tai_Le Tai_Tham Tai_Viet Tamil Telugu Thaana Thai Tibetan Tifinagh Ugaritic Vai Yi

Table 4.8. Mapping of 32-bit character code points to 8-bit units according to UTF-8. The xxx corresponds to the rightmost bit values used in the character code points

Range	Encoding
U-0000 – U-007F	0xxxxxxx
U-0080 – U-07FF	110xxxxx 10xxxxxx
U-0800 – U-FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-010000 – U-10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

```
0046 0045 0302 0054 0045
```

Depending on the operating system, 16-bit codes like U+00CA can be stored with highest byte first – `00CA` – or last – `CA00`. To identify how an operating system orders the bytes of a file, it is a possible to insert a *byte order mark* (BOM), a dummy character tag, at the start of the file. UTF-16 uses the code point U+FEFF to tell whether the storage uses the big-endian convention, where the "big" part of the code is stored first, (`FEFF`) or the little-endian one: (`FFFE`).

UTF-8 is a variable-length encoding. It maps the ASCII code characters U+0000 to U+007F to their byte values 00 to 7F. It then takes on the legacy of ASCII. All the other characters in the range U+007F to U+FFFF are encoded as a sequence of two or more bytes. Table 4.8 shows the mapping principles of the 32-bit character code points to 8-bit units.

Let us encode *FÊTE* in UTF-8. The letters *F*, *T*, and *E* are in the range U-00000000 – U-0000007F. Their numeric code values are exactly the same in ASCII and UTF-8. The code point of *Ê* is `U+00CA` and is in the range U-00000080 – U-000007FF. Its binary representation is 0000 0000 11**00 1010**. UTF-8 uses the 11 rightmost bits of `00CA`. The first five underlined bits together with the prefix 110 form the octet 1100 0011 that corresponds to `C3` in hexadecimal. The seven next boldface bits with the prefix 10 form the octet 10**00 1010** or `8A` in hexadecimal. The letter *Ê* is then encoded as 1100 0011 1000 1010 or `C3 8A` in UTF-8. Hence, the word FÊTE and the code points `U+0046 U+00CA U+0054 U+0045` are encoded as

```
46 C3 8A 54 45
```

UTF-32 represents exactly the codes points by their code values. One question remains: how does UTF-16 represent the code points above U+FFFF? The answer is: it uses two surrogate positions consisting of a high surrogate in the range U+DC00 .. U+DFFF and a low surrogate in the range U+D800 .. U+DBFF. This is made possible because the Unicode consortium does not expect to assign characters beyond the code point U+10FFFF. Using the two surrogates, characters between U+10000 and U+10FFFF can be converted from UTF-32 to UTF-16, and vice versa.

Finally, the storage requirements of the Unicode encoding schemes are, of course, different and depend on the language. A text in English will have approximately the same size in ASCII and in UTF-8. The size of the text will be doubled in UTF-16 and four times its original size in UTF-32, because all characters take four bytes.

A text in a Western European language will be larger in UTF-8 than in ASCII because of the accented characters: a nonaccented character takes one octet, and an accented one takes two. The exact size will thus depend on the proportion of accented characters. The text size will be twice its ASCII size in UTF-16. Characters in the surrogate space take 4 bytes, but they are very rare and should not increase the storage requirements. UTF-8 is then more compact for most European languages. This is not the case with other languages. A Chinese or Indic character takes, on average, three bytes in UTF-8 and only two in UTF-16.

4.3 Locales and Word Order

4.3.1 Presenting Time, Numerical Information, and Ordered Words

In addition to using different sets of characters, languages often have specific presentations for times, dates, numbers, or telephone numbers, even when they are restricted to digits. Most European languages outside English would write $\pi = 3,14159$ instead of $\pi = 3.14159$. Inside a same language, different communities may have different presentation conventions. The US English date February 24, 2003, would be written 24 February 2003 or February 24th, 2003, in England. It would be abridged 2/24/03 in the United States, 24/02/2003 in Britain, and 2003/02/24 in Sweden. Some communities may be restricted to an administration or a company, for instance, the military in the US, which writes times and dates differently than the rest of society.

The International Organization for Standardization (ISO) has standardized the identification of languages and communities under the name of **locales**. Each locale uses a set of rules that defines the format of dates, times, numbers, currency, and how to **collate** – sort – strings of characters. A locale is defined by three parameters: the language, the region, and the variant that corresponds to more specific conventions used by a restricted community. Table 4.9 shows some locales for English, French, and German.

Table 4.9. Examples of locales

Locale	Language	Region	Variant
English (United States)	en	US	
English (United Kingdom)	en	GB	
French (France)	fr	FR	
French (Canada)	fr	CA	
German (Germany)	de	DE	
German (Austria)	de	AT	

Table 4.10. Sorting with the ASCII code comparison and the dictionary order

ASCII order	Dictionary order
ABC	abc
Abc	Abc
Def	ABC
aBf	aBf
abc	def
def	Def

One of the most significant features of a locale is the collation component that defines how to compare and order strings of characters. In effect, elementary sorting algorithms consider the ASCII or Unicode values with a predefined comparison operator such as the inequality predicate < in Python. They determine the lexical order using the numerical ranking of the characters.

These basic sorting procedures do not arrange the words in the classical dictionary order. In ASCII as well as in Unicode, lowercase letters have a greater code value than uppercase ones. A basic algorithm would then sort *above* after *Zambia*, which would be quite misleading for most users.

Current dictionaries in English, French, and German use a different convention. The lowercase letters precede their uppercase equivalents when the strings are equal except for the case. Table 4.10 shows the collation results for some strings.

A basic sorting algorithm may suffice for some applications. However, most of the time it would be unacceptable when the ordered words are presented to a user. The result would be even more confusing with accented characters, since their location is completely random in the extended ASCII tables.

In addition, the lexicographic ordering of words varies from language to language. French and English dictionaries sort accented letters as nonaccented ones, except when two strings are equal except for the accents. Swedish dictionaries treat the letters $\mathring{A}$, $\ddot{A}$, and $\ddot{O}$ as distinct symbols of the alphabet and sort them after *Z*. German dictionaries have two sorting standards. They process accented letters either as single characters or as couples of nonaccented letters. In the latter case, $\ddot{A}$, $\ddot{O}$, $\ddot{U}$, and β are considered respectively as *AE*, *OE*, *UE*, and *ss*.

4.3.2 The Unicode Collation Algorithm

The Unicode consortium has defined a collation algorithm (Davis and Whistler, 2009) that takes into account the different practices and cultures in lexical ordering. It can be parameterized to cover most languages and conventions. It uses three levels of difference to compare strings. We outline their features for European languages and Latin scripts:

- The primary level considers differences between base characters, for instance, between *A* and *B*.
- If there are no differences at the first level, the secondary level considers the accents on the characters.
- And finally, the third level considers the case differences between the characters.

These level features are general, but not universal. Accents are a secondary difference in many languages, but we saw that Swedish sorts accented letters as individual ones and hence sets a primary difference between *A* and *Å*, or *o* and *Ö*. Depending on the language, the levels may have other features.

To deal with the first level, the Unicode collation algorithm defines classes of letters that gather upper- and lowercase variants, accented and unaccented forms. Hence, we have the ordered sets: {a, A, á, Á, à, À, etc.} < {b, B} < {c, C, ć, Ć, ĉ, Ĉ, ç, Ç, etc.} < {e, E, é, É, è, È, ê, Ê, ë, Ë, etc.} < ....

The second level considers the accented letters if two strings are equal at the first level. Accented letters are ranked after their nonaccented counterparts. The first accent is the acute one (´), then come the grave accent (`), the circumflex (ˆ), and the umlaut (¨). So, instances of letter *E* with accents, in lower- and uppercase have the order: {e, E} << {é, É} << {è, È} << {ê, Ê} << {ë, Ë}, where << denotes a difference at the second level. The comparison at the second level is done from the left to the right of a word in English and most languages. It is carried out from the right to the left in French, i.e., from the end of a word to its beginning.

Similarly, the third level considers the case of letters when there are no differences at the first and second levels. Lowercase letters are before uppercase ones, that is, {a} <<< {A}, where <<< denotes a difference at the third level.

Table 4.11 shows the lexical order of *pêcher* 'peach tree' and *Péché* 'sin', together with various conjugated forms of the verbs *pécher* 'to sin' and *pêcher* 'to fish' in French and English. The order takes the three levels into account and the reversed direction of comparison in French for the second level. German adopts the English sorting rules for these accents.

Some characters are expanded or contracted before the comparison. In French, the letters *Œ* and *Æ* are considered as pairs of two distinct letters: *OE* and *AE*. In traditional German used in telephone directories, *Ä*, *Ö*, *Ü*, and *ß* are expanded into *AE*, *OE*, *UE*, and *ss* and are then sorted as an accent difference with the corresponding letter pairs. In traditional Spanish, *Ch* is contracted into a single letter that sorts between *Cz* and *D*.

The implementation of the collation algorithm (Davis and Whistler, 2009, Sect. 4) first maps the characters onto collation elements that have three numerical fields to

Table 4.11. Lexical order of words with accents. Note the reversed order of the second level comparison in French

English	French
Péché	*pèche*
PÉCHÉ	*pêche*
pèche	*Pêche*
pêche	*Péché*
Pêche	*PÉCHÉ*
pêché	*pêché*
Pêché	*Pêché*
pécher	*pécher*
pêcher	*pêcher*

Table 4.12. Some formatting tags in RTF, LaTeX, and HTML

Language	Text in italics	New paragraph	Accented letter *é*
RTF	`{\i text in italics}`	`\par`	`\'e9`
LaTeX	`{\it text in italics}`	`\cr`	`\'{e}`
HTML	`<i>text in italics</i>`	` `	`&eacute;`

express the three different levels of comparison. Each character has constant numerical fields that are defined in a collation element table. The mapping may require a preliminary expansion, as for *æ* and *œ* into *ae* and *oe* or a contraction. The algorithm then forms for each string the sequence of the collation elements of its characters. It creates a sort key by rearranging the elements of the string and concatenating the fields according to the levels: the first fields of the string, then second fields, and third ones together. Finally, the algorithm compares two sort keys using a binary comparison that applies to the first level, to the second level in case of equality, and finally to the third level if levels 1 and 2 show no differences.

4.4 Markup Languages

4.4.1 A Brief Background

Corpus annotation uses sets of labels, also called markup languages. Corpus markup languages are comparable to those of standard word processors such as Microsoft Word or LaTeX. They consist of tags inserted in the text that request, for instance, to start a new paragraph, or to set a phrase in italics or in bold characters. The Rich Text Format (RTF) from Microsoft (2004) and the (La)TeX format designed by Knuth (1986) are widely used markup languages (Table 4.12).

While RTF and LaTeX are used by communities of million of persons, they are not acknowledged as standards. The standard generalized markup language (SGML) takes this place. SGML could have failed and remained a forgotten international

initiative. But the Internet and the World Wide Web, which use hypertext markup language (HTML), a specific implementation of SGML, have ensured its posterity. In the next sections, we introduce the **extensible markup language** (XML), which builds on the simplicity of HTML that has secured its success, and extends it to handle any kind of data.

4.4.2 An Outline of XML

XML is a coding framework: a language to define ways of structuring documents. XML can incorporate logical and presentation markups. Logical markups describe the document structure and organization such as, for instance, the title, the sections, and inside the sections, the paragraphs. Presentation markups describe the text appearance and enable users to set a sentence in italic or bold type, or to insert a page break. Contrary to other markup languages, like HTML, XML does not have a predefined set of tags. The programmer defines them together with their meaning.

XML separates the definition of structure instructions from the content – the data. Structure instructions are described in a document type definition (DTD) that models a class of XML documents. DTDs correspond to specific tagsets that enable users to mark up texts. A DTD lists the legal tags and their relationships with other tags, for instance, to define what is a chapter and to verify that it contains a title. Among coding schemes defined by DTDs, there are:

- the extensible hypertext markup language (XHTML), a clean XML implementation of HTML that models the Internet Web pages;
- the Text Encoding Initiative (TEI), which is used by some academic projects to encode texts, in particular, literary works;
- DocBook, which is used by publishers and open-source projects to produce books and technical documents.

A DTD is composed of three kinds of components called elements, attributes, and entities. Comments of DTDs and XML documents are enclosed between the `<!--` and `-->` tags.

Elements.

Elements are the logical units of an XML document. They are delimited by surrounding tags. A start tag enclosed between angle brackets precedes the element content, and an end tag terminates it. End tags are the same as start tags with a `/` prefix. XML tags must be balanced, which means that an end tag must follow each start tag. Here is a simple example of an XML document inspired by the DocBook specification:

```
<!-- My first XML document -->
<book>
  <title>Language Processing Cookbook</title>
  <author>Pierre Cagné</author>
  <!-- Image to show on the cover -->
```

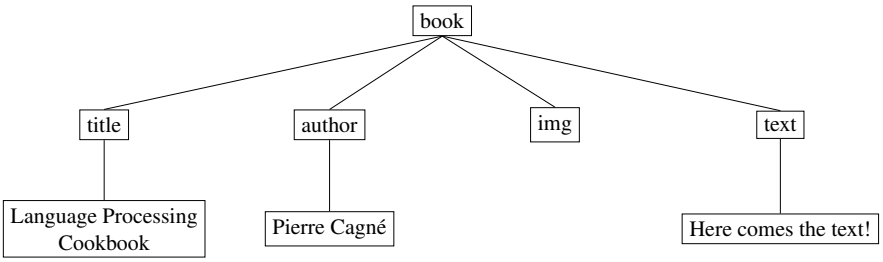

Fig. 4.1. Tree representation of an XML document

```
  <img></img>
  <text>Here comes the text!</text>
</book>
```

where `<book>` and `</book>` are legal tags indicating, respectively, the start and the end of the book, and `<title>` and `</title>` the beginning and the end of the title. **Empty elements**, such as the image `<img></img>`, can be abridged as `<img/>`. Unlike HTML, XML tags are case sensitive: `<TITLE>` and `<title>` define different elements.

We can visualize the structure of an XML document with a parse tree, similar to the syntactic trees we saw for the sentences in Sect. 1.6.1. Figure 4.1 shows the parse tree representing our first XML document.

Attributes.

An element can have attributes, i.e., a set of properties attached to the element. Let us complement our `book` example so that the `<title>` element has an alignment whose possible values are flush left, right, or center, and a character style taken from underlined, bold, or italics. Let us also indicate where `<img>` finds the image file. The DTD specifies the possible attributes of these elements and the value list among which the actual attribute value will be selected. The actual attributes of an element are supplied as name–value pairs in the element start tag.

Let us name the alignment and style attributes `align` and `style` and set them in boldface characters and centered, and let us store the image file name of the `img` element in the `src` attribute. The markup in the XML document will look like:

```
<title align="center" style="bold">
  Language Processing Cookbook
</title>
<author>Pierre Cagné</author>
<img src="pierre.jpg"/>
```

Table 4.13. The predefined entities of XML

Symbol	Entity encoding	Meaning
<	`<`	Less than
>	`>`	Greater than
&	`&`	Ampersand
"	`&quot;`	Quotation mark
’	`&apos;`	Apostrophe

Entities.

Finally, entities correspond to data stored somewhere in a computer. They can be accented characters, symbols, strings as well as text or image files. The programmer declares or defines variables referring to entities in a DTD and uses them subsequently in XML documents. There are two types of entities: general and parameter. General entities, or simply entities, are declared in a DTD and used in XML document contents. Parameter entities are only used in DTDs. The two types of entities correspond to two different contexts. They are declared and referred to differently. We set aside the parameter entities here; we will examine them in the next section.

An entity is referred to within an XML document by enclosing its name between the start delimiter "`&`" and the end delimiter "`;`", such as `&EntityName;`. The XML parser will substitute the reference with the content of `EntityName` when it is encountered.

XML recognizes a set of predefined or implicitly defined entities that do not need to be declared in a DTD. These entities are used to encode special or accented characters. They can be divided into two groups. The first group consists of five predefined entities (Table 4.13). They correspond to characters used by the XML standard, which cannot be used as is in a document. The second group, called numeric character entities, is used to insert non-ASCII symbols or characters. Character references consist of a Unicode hexadecimal number delimited by "`&#x`" and "`;`", such as `&#xc9;` for *É* and `&#xA9;` for ©.

4.4.3 Writing a DTD

The DTD specifies the formal structure of a document type. It enables an XML parser to determine whether a document is valid. The DTD file contains the description of all the legal elements, attributes, and entities.

Elements.

The description of the elements is enclosed between the start and end delimiters `<!ELEMENT` and `>`. It contains the element name and the content model in terms of other elements or reserved keywords (Table 4.14). The content model specifies how the elements appear, their order, and their number of occurrences (Table 4.15). For example:

Table 4.14. Character types

Character type	Description
`PCDATA`	Parsed character data. This data will be parsed and must only be text, punctuation, and special characters; no embedded elements
`ANY`	`PCDATA` or any DTD element
`EMPTY`	No content – just a placeholder

Table 4.15. List separators and occurrence indicators

List notation	Description
`,`	Elements must all appear and be ordered as listed
`|`	Only one element must appear (exclusive or)
`+`	Compulsory element (one or more)
`?`	Optional element (zero or one)
`*`	Optional element (zero or more)

```
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
```

states that a `book` consists of a `title`, a possible `author` or `editor`, an image `img`, and one or more `chapter`s. The `title` consists of `PCDATA`, that is, only text with no other embedded elements.

Attributes.

Attributes are the possible properties of the elements. Attribute lists are usually defined after the element they refer to. Their description is enclosed between the delimiters `<!ATTLIST` and `>`. An attribute list contains:

- the element the attribute refers to
- the attribute name
- the kind of value the attribute may take: a predefined type (Table 4.16) or an enumerated list of values between brackets and separated by vertical bars
- the default value between quotes or a predefined keyword (Table 4.17)

For example:

```
<!ATTLIST title
  style (underlined | bold | italics) "bold"
  align (left | center | right) "left">
<!ATTLIST author
  style (underlined | bold | italics) #REQUIRED>
```

says that `title` has two attributes, `style` and `align`. The `style` attribute can have three possible values and, if not specified in the XML document, the default value will be `bold`; `author` has one `style` attribute that must be specified in the document.

Table 4.16. Some XML attribute types

Attribute types	Description
`CDATA`	The string type: any character except `<`, `>`, `&`, `'`, and `"`
`ID`	An identifier of the element unique in the document; `ID` must begin with a letter, an underscore, or a colon
`IDREF`	A reference to an identifier
`NMTOKEN`	String of letters, digits, periods, underscores, hyphens, and colons. It is more restrictive than `CDATA`; for instance, spaces are not allowed

Table 4.17. Some default value keywords

Predefined default values	Description
`#REQUIRED`	A value must be supplied
`#FIXED`	The attribute value is constant and must be equal to the default value
`#IMPLIED`	If no value is supplied, the processing system will define the value

Entities.

Entities enable users to define variables in a DTD. Their declaration is enclosed between the delimiters `<!ENTITY` and `>`. It contains the entity name and the entity content (possibly a sequence):

```
<!ENTITY myEntity "Introduction">
```

This entity can then be used in an XML document with the reference `&myEntity;`. The XML parser will replace all the references it encounters with the value `Introduction`.

Parameter entities are only used in DTDs. They have a "`%`" sign before the entity name, as in

```
<!ENTITY % myParEntity "<!ELEMENT textbody (para)+>">
```

Further references to parameter entities in a DTD use "`%`" and "`;`" as delimiters, such as `%myParEntity;`.

A DTD Example.

Let us now suppose that we want to publish cookbooks. We define a document type, and we declare the rules that will form its DTD: a book will consist of a title, a possible author or editor, an image, one or more chapters, and one or more paragraphs in these chapters. Let us then suppose that the main title and the chapter titles can be in bold, in italics, or underlined. Let us finally suppose that the chapter titles can be numbered in Roman or Arabic notation. The DTD elements and attributes are

```
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
<!ATTLIST title style (u | b | i) "b">
<!ELEMENT author (#PCDATA)>
<!ATTLIST author style (u | b | i) "i">
<!ELEMENT editor (#PCDATA)>
<!ATTLIST editor style (u | b | i) "i">
<!ELEMENT img EMPTY>
<!ATTLIST img src CDATA #REQUIRED>
<!ELEMENT chapter (subtitle, para+)>
<!ATTLIST chapter number ID #REQUIRED>
<!ATTLIST chapter numberStyle (Arabic | Roman) "Roman">
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT para (#PCDATA)>
```

The name of the document type corresponds to the **root element**, here `book`, which must be unique.

XML Schema.

You probably noticed that the DTD syntax does not fit very well with that of XML. This bothered some people, who tried to make it more compliant. This gave birth to XML Schema, a document definition standard using the XML style. As of today, DTD is still "king," however, XML Schema is gaining popularity. Specifications are available from the Web consortium at `http://www.w3.org/XML/Schema`.

4.4.4 Writing an XML Document

We shall now write a document conforming to the `book` document type. A complete XML document begins with a prologue, a declaration like this one:

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
```

describing the XML version, the encoding used, and whether the document is self-contained or not (standalone). In our example, if we have an external DTD, we must set `standalone` to `no`. This prologue is mandatory from version 1.1 of XML. If not specified, the default encoding is UTF-8.

The document can contain any Unicode character. The encoding refers to how the characters are stored in the file. This has no significance if you only use unaccented characters in the basic Latin set from position 0 to 127. If you type accented characters, the editor will have to save them as UTF-8 codes. In the document above, *Cagné* must be stored as 43 61 67 6E C3 A9, where *é* is corresponds to C3 A9.

If your text editor does not support UTF-8, you will have to enter the accented characters as entities with their Unicode code point, for instance, `&#xC9;` for *É*, or `&#xE9;` for *é*. You may also type the characters *É* or *é* and use your machine's default encoding, such as Latin 1 (ISO-8859-1), Windows-1252, or MacRoman, to save the

XML file. You will have then to declare the corresponding encoding, for instance, `encoding="ISO-8859-1"`.

Then, the document declares the DTD it uses. The DTD can be inside the XML document and enclosed between the delimiters `<!DOCTYPE [` and `]>`, for instance:

```
<!DOCTYPE book [
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
...
]>
```

Or the DTD can be external to the document, for instance, in a file called `book_definition.dtd`. In this case, `DOCTYPE` indicates its location on the computer using the keyword `SYSTEM`:

```
<!DOCTYPE book SYSTEM "/home/pierre/xml/book_definition.dtd">
```

Finally, we can write the document content. Let us use the XML tags to sketch a very short book. It could look like this:

```
<book>
  <title style="i">Language Processing Cookbook</title>
  <author style="b">Pierre Cagné</author>
  <img src="pierre.jpg"/>
  <chapter number="c1">
    <subtitle>Introduction</subtitle>
    <para>Let&apos;s start doing simple things:
      Collect texts.
    </para>
    <para>First, choose an author you like.</para>
  </chapter>
</book>
```

with a graphical representation in Fig. 4.2.

Once, we have written an XML document, we must check that is **well formed**, which means that it has no syntax errors: the brackets are balanced, the encoding is correct, etc. We must also **validate** it, i.e., check that it conforms to the DTD. This can be done with a variety of parsers available from the Internet, for instance the W3C markup validation service (`http://validator.w3.org/`). Another easy way to do it is to use the embedded XML parser of any a modern web browser.

4.4.5 Namespaces

In our examples, we used element names that can be part of other DTDs. The string `title`, for instance, is used by XHTML. The XML namespaces is a device to avoid collisions. It is a naming scheme that enables us to define groups of elements and attributes in the same document and prevent name conflicts.

We declare a namespace using the predefined `xmlns` attribute as

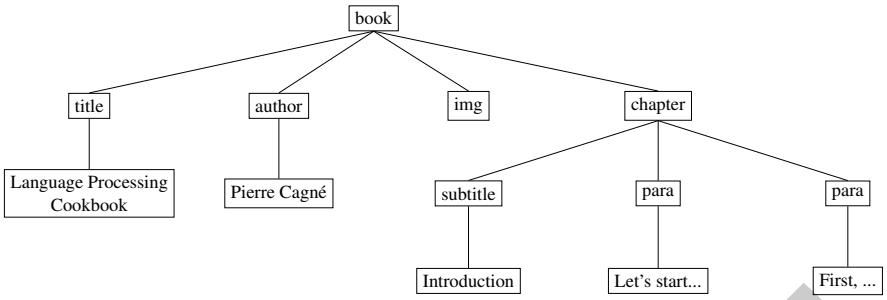

Fig. 4.2. Tree representation of the document

```
<my-element xmlns:prefix="URI">
```

It starts a namespace inside `my-element` and its descendants, where `prefix` defines a group of names. Names members of this namespace are preceded by the prefix, as in `prefix:title`. `URI` has the syntax of a web address. However, it is just a unique name; it is never accessed.

Declaring two namespaces in `book`, we can reuse `title` for different purposes:

```
<book
  xmlns:pierre="http://www.cs.lth.se/~pierre"
  xmlns:raymond="http://www.grandecuisine.com">

  <pierre:title style="i">Language Processing Cookbook
  </pierre:title>

  <raymond:title style="i">A French Cookbook
  </raymond:title>
</book>
```

4.4.6 XML and Databases

Although we introduced XML to annotate corpora and narrative documents, many applications use it to store and exchange structured data like records, databases, or configuration files. In fact, creating tabular data in the form of collections of property names and values is easy with XML: we just need to define elements to mark the names (or keys) and the values. Such structures are called dictionaries, like this one:

```
<dict>
  <key>language</key>  <value>German</value>
  <key>currency</key>  <value>euro</value>
</dict>
```

As soon as it was created, XML gained a large popularity among program developers for this purpose. People found it easier to use XML rather than creating

their own solution because of its simplicity, its portability, and the wide availability of parsers.

4.5 Collecting Corpora from the Web

While collecting corpora used to be a relatively tedious operation, the advent of the Web has turned it into a child's play. The Web is now the host of zillions of documents that just wait for being fetched, a process also called *scraping*.

4.5.1 Scraping Documents with Python

Python has a set of modules that enables a programmer to download the content of a web page given its address, a URL. To get the Wikipedia page on Aristotle, we only need a 3-line program:

```
import requests

url_en = 'https://en.wikipedia.org/wiki/Aristotle'
html_doc = requests.get(url_fr).text
```

where we use the `requests` module, `get()` to open and read the page, and the `text` attribute to access the HTML document.

Printing the `html_doc` variable shows the page with all its markup:

```
<!DOCTYPE html>
<html lang="en" dir="ltr" class="client-nojs">
<head>
<meta charset="UTF-8"/>
<title>Aristotle - Wikipedia, the free encyclopedia</title>
...
```

4.5.2 HTML

HTML is a kind of XML with specific elements like `<title>` to markup a title, `<body>` for the body of a page, `<h1>`, `<h2>`, ..., `<h6>`, for headings from the highest level to the lowest one, `<p>` for a paragraph, etc. Although, it is no longer the case, HTML used to be defined with a DTD.

One of the most important features of HTML is its ability to define links to any page of the Web through hyperlinks. We create such hyperlinks using the `<a>` element (anchor) and its `href` attribute as in:

```
<a href="https://en.wikisource.org/wiki/Author:Aristotle">
  Wikisource</a>
```

where the text inside the start and end `<a>` tags, *Wikisource*, will show in blue on the page and will be sensitive to user interaction. In the context of Wikipedia, this text in blue is called a label or, in the Web jargon, an anchor. If the user clicks on it, s/he will be moved to the page stored in the `href` attribute: `https://en.wikisource.org/wiki/Author:Aristotle`. This part is called the link or the target. In our case, it will lead to Wikisource, a library of free texts, and here to the works of Aristotle translated in English.

4.5.3 Parsing HTML

Before we can apply language processing components to a page collected from the Web, we need to parse its HTML structure and extract the data. To carry this out, we will use Beautiful Soup, a very popular HTML parser (`https://www.crummy.com/software/BeautifulSoup/`).

As in the previous section, we fetch a document using the `requests` module. We then parse it with Beautiful Soup, where we need to tell which HTML parser to use. In the code below, we use the standard Python parser `html.parser`:

```
import bs4
import requests

url_en = 'https://en.wikipedia.org/wiki/Aristotle'
html_doc = requests.get(url_en).text
parse_tree = bs4.BeautifulSoup(html_doc, 'html.parser')
```

The `parse_tree` variable contains the parsed HTML document from which we can access its elements and their attributes. We access the title and its markup through the title attribute of `parse_tree` (`parse_tree.title`) and the content of the title with `parse_tree.title.text`:

```
parse_tree.title
# <title>Aristotle - Wikipedia, the free encyclopedia</title>
parse_tree.title.text
# Aristotle - Wikipedia, the free encyclopedia
```

The first heading `h1` corresponds to the title of the article,

```
parse_tree.h1.text
# Aristotle
```

while the `h2` headings contain its subtitles. We access the list of subtitles using the `find_all()` method:

```
headings = parse_tree.find_all('h2')
[heading.text for heading in headings]
# ['Contents', 'Life', 'Thought', 'Loss and preservation of
his works', 'Legacy', 'List of works', 'Eponyms', 'See also',
'Notes and references', 'Further reading', 'External links',
'Navigation menu']
```

Finally, we can easily find all the links and the labels from a Web page with this statement:

```
links = parse_tree.find_all('a', href=True)
```

where we collect all the anchors that have a `href` attribute. Then, we create the list of labels:

```
[link.text for link in links]
```

and the list of links using the `get()` method:

```
[link.get('href') for link in links]
```

which will return `None` if there is no link. Alternatively, we can find the links with a dictionary notation:

```
[link['href'] for link in links]
```

This will raise a `KeyError` if there is no link. This should not happen here as we specified `href=True` when we collected the links.

The Web addresses (URL) can either be absolute i.e. containing the full address including the host name like `https://en.wikipedia.org/wiki/Aristotle`, or relative to the start page with just the file name, like `/wiki/Organon`. If we need to access the latter Web page, we need to create an absolute address, `https://en.wikipedia.org/wiki/Organon`, from the relative one. We can do this with the `urljoin()` function and this program:

```
from urllib.parse import urljoin

url_en = 'https://en.wikipedia.org/wiki/Aristotle'
...
[urljoin(url_en, link['href']) for link in links]
# List of absolute addresses
```

4.6 Further Reading

Many operating systems such as Windows, Mac OS X, and Unix, or programming languages such as Java have adopted Unicode and take the language parameter of a computer into account. Basic lexical methods such as date and currency formatting, word ordering, and indexing are now supported at the operating system level. Operating systems or programming languages offer toolboxes and routines that you can use in applications.

The Unicode Consortium publishes books, specifications, and technical reports that describe the various aspects of the standard. *The Unicode Standard* (The Unicode Consortium, 2012) is the most comprehensive document, while Davis and Whistler (2009) describe in detail the Unicode collation algorithm. Both documents are available in electronic format from the Unicode web site: `http://www.`

unicode.org/. The Unicode Consortium also maintains a public and up-to-date version of the character database (http://www.unicode.org/ucd/). IBM implemented a large library of Unicode components in Java and C++, which are available as open-source software (http://site.icu-project.org/).

HTML and XML markup standards are continuously evolving. Their specifications are available from the World Wide Web consortium (http://www.w3.org/). HTML and XML parsers are available for most programming languages. Beautiful Soup is the most popular one for Python and has an excellent documentation (https://www.crummy.com/software/BeautifulSoup/). Finally, a good reference on XML is *Learning XML* (Ray, 2003).

Exercises

4.1. Implement UTF-8 that transforms a sequence of code points in a sequence of octets in Python.

4.2. Implement a word collation algorithm for English, French, German, or Swedish.

4.3. Modify the DTD in Sect. 4.4.4 so that the cookbook consists of meals instead of chapters, and each meal has an ingredient and a recipe section.

4.4. Modify the DTD in Sect. 4.4.4 to declare the general and parameter entities:

```
<!ENTITY myEntity "Introduction">
<!ENTITY %myEntity "<!ELEMENT textbody (para)+>">
```

Use these entities in the DTD and the document.

4.5. Write a Python program that removes the tags from a text encoded in HTML.

4.6. Write a Python program that processes a text encoded in HTML: it retains headers (`Hn` tags) and discards the rest.

4.7. Write a Python program that connects to a web site, and explore hypertext web links using a breadth-first strategy.

5

Topics in Information Theory and Machine Learning

5.1 Introduction

Information theory underlies the design of codes. Claude Shannon probably started the field with a seminal article (1948), in which he defined a measure of information: the **entropy**. In this chapter, we introduce essential concepts in information theory: entropy, optimal coding, cross entropy, and **perplexity**. Entropy is a very versatile measure of the average information content of symbol sequences and we will explore how it can help us design efficient encodings.

In natural language processing, we often need to determine the category of an object or an observation, such as the part of speech of a word. We will show how we can use entropy to learn decision trees from data sets. This will enable us to build a simple and essential machine-learning algorithm: ID3. We will apply the decision trees we derive from the data sets as classifiers, i.e., devices to classify new data or new objects.

Machine-learning techniques are now instrumental in most areas of natural language processing, and we will use them throughout this book. We will conclude this chapter with the description of three other linear classifiers from among the most popular ones.

5.2 Codes and Information Theory

5.2.1 Entropy

Information theory models a text as a sequence of symbols. Let $x_1, x_2, ..., x_N$ be a discrete set of N symbols representing the characters. The **information content** of a symbol is defined as

$$I(x_i) = -\log_2 P(x_i) = \log_2 \frac{1}{P(x_i)},$$

Table 5.1. Letter frequencies in the French novel *Salammbô* by Gustave Flaubert. The text has been normalized in uppercase letters. The table does not show the frequencies of the punctuation signs or digits

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
A	42,439	*L*	30,960	*W*	1	*Ë*	6
B	5,757	*M*	13,090	*X*	2,206	*Î*	277
C	14,202	*N*	32,911	*Y*	1,232	*Ï*	66
D	18,907	*O*	22,647	*Z*	413	*Ô*	397
E	71,186	*P*	13,161	*À*	1,884	*Œ*	96
F	4,993	*Q*	3,964	*Â*	605	*Ù*	179
G	5,148	*R*	33,555	*Æ*	9	*Û*	213
H	5,293	*S*	46,753	*Ç*	452	*Ü*	0
I	33,627	*T*	35,084	*É*	7,709	*Ÿ*	0
J	1,220	*U*	29,268	*È*	2,002	Blanks	103,481
K	92	*V*	6,916	*Ê*	898	**Total:**	593,299

and it is measured in bits. When the symbols have equal probabilities, they are said to be equiprobable and

$$P(x_1) = P(x_2) = ... = P(x_N) = \frac{1}{N}.$$

The information content of x_i is then $I(x_i) = \log_2 N$.

The information content corresponds to the number of bits that are necessary to encode the set of symbols. The information content of the alphabet, assuming that it consists of 26 unaccented equiprobable characters and the space, is $\log_2(26+1) = 4.75$, which means that 5 bits are necessary to encode it. If we add 16 accented characters, the uppercase letters, 11 punctuation signs, [, . ; : ? ! " – ( ) ’], and the space, we need $(26+16)\times 2+12 = 96$ symbols. Their information content is $\log_2 96 = 6.58$, and they can be encoded on 7 bits.

The information content assumes that the symbols have an equal probability. This is rarely the case in reality. Therefore this measure can be improved using the concept of entropy, the average information content, which is defined as:

$$H(X) = -\sum_{x\in X} P(x)\log_2 P(x),$$

where X is a random variable over a discrete set of variables, $P(x) = P(X = x), x \in X$, with the convention $0\log_2 0 = 0$. When the symbols are equiprobable, $H(X) = \log_2 N$. This also corresponds to the upper bound on the entropy value, and for any random variable, we have the inequality $H(X) \leq \log_2 N$.

To evaluate the entropy of printed French, we computed the frequency of the printable French characters in Gustave Flaubert’s novel *Salammbô*. Table 5.1 shows the frequency of 26 unaccented letters, the 16 accented or specific letters, and the blanks (spaces).

Table 5.2. Frequency counts of the symbols

	A	B	C	D	E	F	G	H
Freq	42,439	5,757	14,202	18,907	71,186	4,993	5,148	5,293
Prob	0.25	0.03	0.08	0.11	0.42	0.03	0.03	0.03

Table 5.3. A possible encoding of the symbols on 3 bits

A	B	C	D	E	F	G	H
000	001	010	011	100	101	110	111

The entropy of the text restricted to the characters in Table 5.1 is defined as:

$$\begin{aligned} H(X) &= -\sum_{x\in X} P(x)\log_2 P(x). \\ &= -P(A)\log_2 P(A) - P(B)\log_2 P(B) - P(C)\log_2 P(C) - ... \\ &\quad -P(Z)\log_2 P(Z) - P(\grave{A})\log_2 P(\grave{A}) - P(\hat{A})\log_2 P(\hat{A}) - ... \\ &\quad -P(\ddot{U})\log_2 P(\ddot{U}) - P(\ddot{Y})\log_2 P(\ddot{Y}) - P(\textit{blanks})\log_2 P(\textit{blanks}). \end{aligned}$$

If we distinguish between upper- and lowercase letters and if we include the punctuation signs, the digits, and all the other printable characters – ASCII ≥ 32 – the entropy of Gustave Flaubert's *Salammbô* in French is $H(X) = 4.376$.

5.2.2 Huffman Coding

The information content of the French character set is less than the 7 bits required by equiprobable symbols. Although it gives no clue about an encoding algorithm, it indicates that a more efficient code is theoretically possible. This is what we examine now with Huffman coding, which is a general and simple method to build such a code.

Huffman coding uses variable-length code units. Let us simplify the problem and use only the eight symbols A, B, C, D, E, F, G, and H with the count frequencies in Table 5.2.

The information content of equiprobable symbols is $\log_2 8 = 3$ bits. Table 5.3 shows a possible code with constant-length units.

The idea of Huffman coding is to encode frequent symbols using short code values and rare ones using longer units. This was also the idea of the Morse code, which assigns a single signal to letter E: `.`, and four signals to letter X: `-..-`.

This first step builds a Huffman tree using the frequency counts. The symbols and their frequencies are the leaves of the tree. We grow the tree recursively from the leaves to the root. We merge the two symbols with the lowest frequencies into a new node that we annotate with the sum of their frequencies. In Fig. 5.1, this new node corresponds to the letters F and G with a combined frequency of 4993 + 5148 = 10,141 (Fig. 5.2). The second iteration merges B and H (Fig. 5.3); the third one, (F,G) and (B,H) (Fig. 5.4), and so on (Figs. 5.5–5.8).

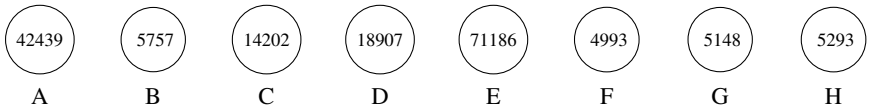

Fig. 5.1. The symbols and their frequencies

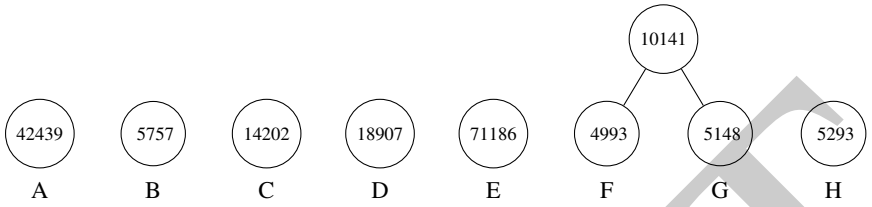

Fig. 5.2. Merging the symbols with the lowest frequencies

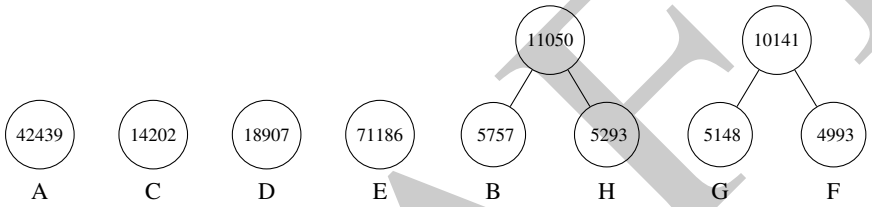

Fig. 5.3. The second iteration

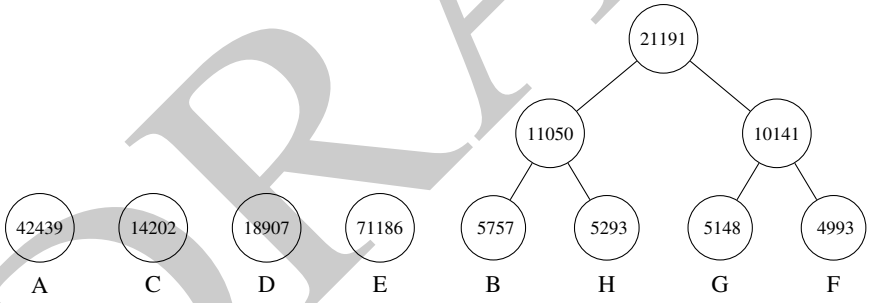

Fig. 5.4. The third iteration

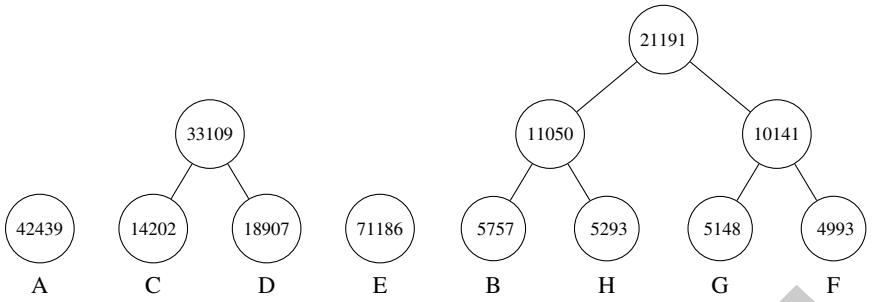

Fig. 5.5. The fourth iteration

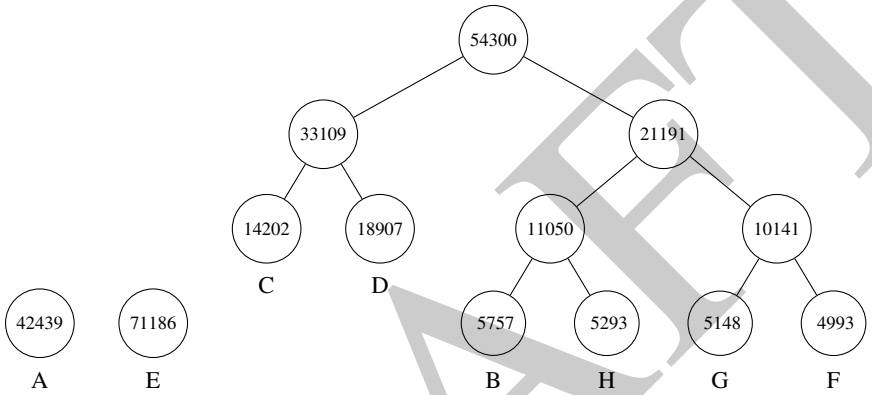

Fig. 5.6. The fifth iteration

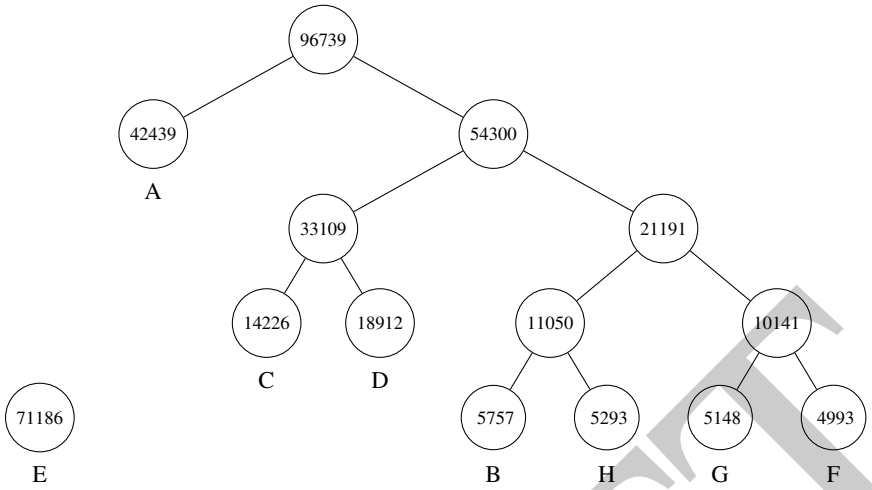

Fig. 5.7. The sixth iteration

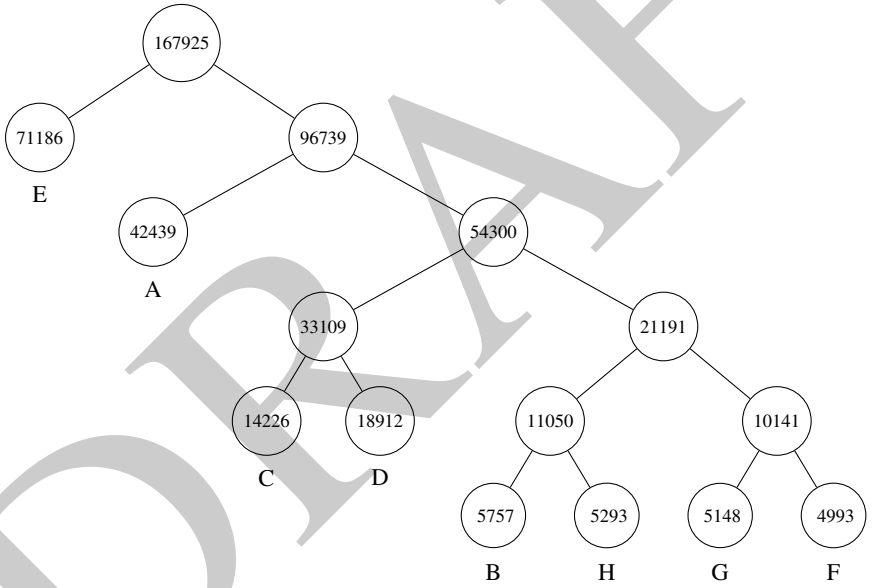

Fig. 5.8. The final Huffman tree

Table 5.4. The Huffman code

A	B	C	D	E	F	G	H
10	11100	1100	1101	0	11111	11110	11101

The second step of the algorithm generates the Huffman code by assigning a 0 to the left branches and a 1 to the right branches (Table 5.4).

The average number of bits is the weighted length of a symbol. If we compute it for the data in Table 5.2, it corresponds to:

$$0.25 \times 2 \text{ bit} + 0.03 \times 5 \text{ bit} + 0.08 \times 4 \text{ bit} + 0.11 \times 4 \text{ bit} + 0.42 \times 1 \text{ bit}$$
$$+0.03 \times 5 \text{ bit} + 0.03 \times 5 \text{ bit} + 0.03 \times 5 \text{ bit} = 2.35$$

We can compute the entropy from the counts in Table 5.2. It is defined by the expression:

$$-\left(\frac{42439}{167925} \log_2 \frac{42439}{167925} + \frac{5757}{167925} \log_2 \frac{5757}{167925} + \frac{14202}{167925} \log_2 \frac{14202}{167925} + \right.$$
$$\frac{18907}{167925} \log_2 \frac{18907}{167925} + \frac{71186}{167925} \log_2 \frac{71186}{167925} + \frac{4993}{167925} \log_2 \frac{4993}{167925} +$$
$$\left. \frac{5148}{167925} \log_2 \frac{5148}{167925} + \frac{5293}{167925} \log_2 \frac{5293}{167925} \right) = 2.31$$

We can see that although the Huffman code reduces the average number of bits from 3 to 2.35, it does not reach the limit defined by entropy, which is, in our example, 2.31.

5.2.3 Cross Entropy

Let us now compare the letter frequencies between two parts of *Salammbô*, then between *Salammbô* and another text in French or in English. The symbol probabilities will certainly be different. Intuitively, the distributions of two parts of the same novel are likely to be close, further apart between *Salammbô* and another French text from the 21st century, and even further apart with a text in English. This is the idea of cross entropy, which compares two probability distributions.

In the cross entropy formula, one distribution is referred to as the model. It corresponds to data on which the probabilities have been trained. Let us name it M with the distribution $M(x_1), M(x_2), ..., M(x_N)$. The other distribution, P, corresponds to the test data: $P(x_1), P(x_2), ..., P(x_N)$. The cross entropy of M on P is defined as:

$$H(P,M) = -\sum_{x \in X} P(x) \log_2 M(x).$$

Cross entropy quantifies the average surprise of the distribution when exposed to the model. We have the inequality $H(P) \leq H(P,M)$ for any other distribution M with

Table 5.5. The entropy is measured on the file itself and the cross entropy is measured with Chapters 1–14 of Gustave Flaubert's *Salammbô* taken as the model

	Entropy	**Cross entropy**	**Difference**
Salammbô, chapters 1-14, training set	4.37745	4.37745	0.0
Salammbô, chapter 15, test set	4.31793	4.33003	0.01210
Notre Dame de Paris, test set	4.43696	4.45590	0.01894
Nineteen Eighty-Four, test set	4.35922	4.80767	0.44845

Table 5.6. The perplexity and cross perplexity of texts measured with Chapters 1–14 of Gustave Flaubert's *Salammbô* taken as the model

	Perplexity	**Cross perplexity**
Salammbô, chapters 1-14, training set	20.78	20.78
Salammbô, chapter 15, test set	19.94	20.11
Notre Dame de Paris, test set	21.66	21.95
Nineteen Eighty-Four, test set	20.52	28.01

equality if and only if $M(x_i) = P(x_i)$ for all i. The difference, also called the Kullback and Leibler (1951) divergence,

$$D_{KL}(P||M) = H(P,M) - H(P)$$

is a measure of the relevance of the model: the closer the cross entropy, the better the model.

To see how the probability distribution of Flaubert's novel could fare on other texts, we trained a model on the first fourteen chapters of *Salammbô*, and we applied it to the last chapter of *Salammbô* (Chap. 15), to Victor Hugo's *Notre Dame de Paris*, both in French, and to *Nineteen Eighty-Four* by George Orwell in English. The data in Table 5.5 conform to our intuition. They show that the first chapters of *Salammbô* are a better model of the last chapter of *Salammbô* than of *Notre Dame de Paris*, and even better than of *Nineteen Eighty-Four*.

5.2.4 Perplexity and Cross Perplexity

Perplexity is an alternate measure of information that is mainly used by the speech processing community. Perplexity is simply defined as $2^{H(X)}$. The cross perplexity is defined similarly as $2^{H(P,M)}$.

Although perplexity does not bring anything new to entropy, it presents the information differently. Perplexity reflects the averaged number of choices of a random variable. It is equivalent to the size of an imaginary set of equiprobable symbols, which is probably easier to understand.

Table 5.6 shows the perplexity and cross perplexity of the same texts measured with Chaps. 1–14 of Gustave Flaubert's *Salammbô* taken as the model.

Table 5.7. A set of object members of two classes: N and P. Here the objects are weather observations. After Quinlan (1986)

Object	Attributes				Class
	Outlook	**Temperature**	**Humidity**	**Windy**	
1	Sunny	Hot	High	False	N
2	Sunny	Hot	High	True	N
3	Overcast	Hot	High	False	P
4	Rain	Mild	High	False	P
5	Rain	Cool	Normal	False	P
6	Rain	Cool	Normal	True	N
7	Overcast	Cool	Normal	True	P
8	Sunny	Mild	High	False	N
9	Sunny	Cool	Normal	False	P
10	Rain	Mild	Normal	False	P
11	Sunny	Mild	Normal	True	P
12	Overcast	Mild	High	True	P
13	Overcast	Hot	Normal	False	P
14	Rain	Mild	High	True	N

5.3 Entropy and Decision Trees

Decision trees are useful devices to classify objects into a set of classes. In this section, we describe what they are and see how entropy can help us learn – or induce – automatically decision trees from a set of data. The algorithm, which resembles a reverse Huffman encoding, is one of the simplest machine-learning techniques.

5.3.1 Machine Learning

Machine learning considers collections of objects or observations, where each object is defined by a set of attributes, SA. Each attribute has a set of possible values called the attribute domain. Table 5.7 from Quinlan (1986) shows a collection of objects, where:

$$SA = \{Outlook, Temperature, Humidity, Windy\},$$

with the following respective domains:

- $\mathrm{dom}(Outlook) = \{sunny, overcast, rain\}$,
- $\mathrm{dom}(Temperature) = \{hot, mild, cool\}$,
- $\mathrm{dom}(Humidity) = \{normal, high\}$,
- $\mathrm{dom}(Windy) = \{true, false\}$.

Each object is member of a class, N or P in this data set.

Machine-learning algorithms can be categorized along two main lines: supervised and unsupervised classification. In supervised machine-learning, each object belongs to a predefined class, here P and N. This is the technique we will use in the induction of decision trees, where we will automatically create a tree from a training

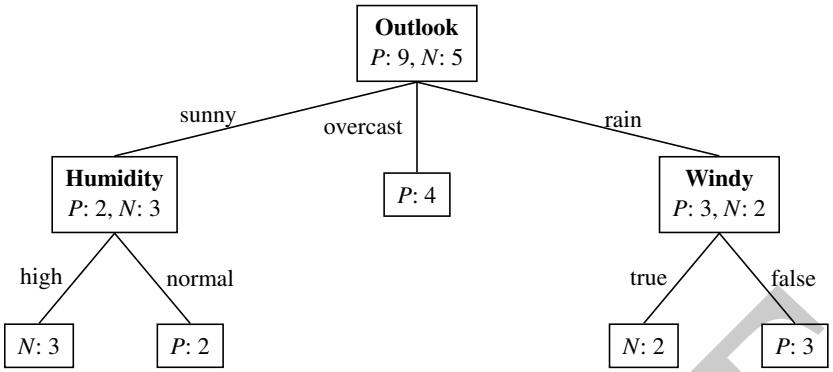

Fig. 5.9. A decision tree classifying the objects in Table 5.7. Each node represents an attribute with the number of objects in the classes *P* and *N*. At the start of the process, the collection has nine objects in class *P* and five in class *N*. The classification is done by testing the attribute values of each object in the nodes until a leaf is reached, where all the objects belong to one class, *P* or *N*. After Quinlan (1986)

set, here the examples in Fig. 5.7. Once the tree is induced, it will be able to predict the class of examples taken outside the training set.

Machine-learning techniques make it possible to build programs that organize and classify data, like annotated corpora, without the chore of manually explicating the rules behind this organization or classification. Because of the availability of massive volumes of data, they have become extremely popular in all the fields of language processing. They are now instrumental in many NLP applications and tasks, including part-of-speech tagging, parsing, semantic role labeling, or coreference solving, that we will describe in the next chapters of this book.

5.3.2 Decision Trees

A decision tree is a tool to classify objects such as those in Table 5.7. The nodes of a tree represent conditions on the attributes of an object, and a node has as many branches as its corresponding attribute has values. An object is presented at the root of the tree, and the values of its attributes are tested by the tree nodes from the root down to a leaf. The leaves return a decision, which is the object class or probabilities to be the member of a class.

Fig. 5.9 shows a decision tree that correctly classifies all the objects in the set shown in Table 5.7 (Quinlan, 1986).

5.3.3 Inducing Decision Trees Automatically

It is possible to design many trees that classify successfully the objects in Table 5.7. The tree in Fig. 5.9 is interesting because it is efficient: a decision can be made with a minimal number of tests.

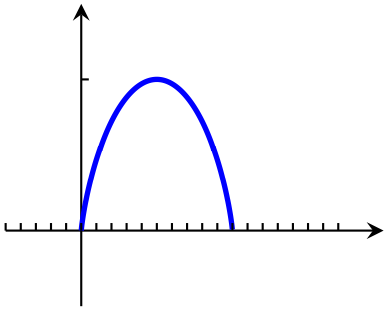

Fig. 5.10. The binary entropy function: $-x\log_2 x-(1-x)\log_2(1-x)$

An efficient decision tree can be induced from a set of examples, members of mutually exclusive classes, using an entropy measure. We will describe the induction algorithm using two classes of p positive and n negative examples, although it can be generalized to any number of classes. As we saw earlier, each example is defined by a finite set of attributes, SA.

At the root of the tree, the condition, and hence the attribute, must be the most discriminating, that is, have branches gathering most positive examples while others gather negative examples. A perfect attribute for the root would create a partition with subsets containing only positive or negative examples. The decision would then be made with one single test. The ID3 (Quinlan, 1986) algorithm uses this idea and the entropy to select the best attribute to be this root. Once we have the root, the initial set is split into subsets according to the branching conditions that correspond to the values of the root attribute. Then, the algorithm determines recursively the next attributes of the resulting nodes.

ID3 defines the information gain of an attribute as the difference of entropy before and after the decision. It measures its separating power: the more the gain, the better the attribute. At the root, the entropy of the collection is constant. As defined previously, for a two-class set of p positive and n negative examples, it is:

$$I(p,n) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}.$$

Figure 5.10 shows this binary entropy function with $x = \dfrac{p}{p+n}$, for x ranging from 0 to 1. The function attains its maximum of 1 at $x = 0.5$, when $p = n$ and there are as many positive as negative examples in the set, and its minimum of 0 at $x = 0$ and $x = 1$, when $p = 0$ or $n = 0$ and the examples in the set are either all positive or all negative.

An attribute A with v possible values $\{A_1, A_2, ..., A_v\}$ creates a partition of the collection into v subsets, where each subset corresponds to one value of A and contains p_i positive and n_i negative examples. The entropy of a subset is $I(p_i, n_i)$ and the weighted average of entropies of the partition created by A is:

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}).$$

The information gain is defined as $Gain(A) = I(p,n) - E(A)$ (or $I_{\text{before}} - I_{\text{after}}$). We would reach the maximum possible gain with an attribute that creates subsets containing examples that are either all positive or all negative. In this case, the entropy of the nodes below the root would be 0.

For the tree in Fig. 5.9, let us compute the information gain of attribute *Outlook*. The entropy of the complete data set is:

$$I(p,n) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940.$$

Outlook has three values: *sunny*, *overcast*, and *rain*. The entropies of the respective subsets created by these values are:

$$\begin{aligned} &\textit{sunny}: && I(p_1,n_1) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.971. \\ &\textit{overcast}: && I(p_2,n_2) = 0. \\ &\textit{rain}: && I(p_3,n_3) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.971. \end{aligned}$$

Thus

$$E(Outlook) = \frac{5}{14}I(p_1,n_1) + \frac{4}{14}I(p_2,n_2) + \frac{5}{14}I(p_3,n_3) = 0.694.$$

$Gain(Outlook)$ is then 0.940 – 0.694 = 0.246, which is the highest for the four attributes. $Gain(Temperature)$, $Gain(Humidity)$, and $Gain(Windy)$ are computed similarly.

The algorithm to build the decision tree is simple. The information gain is computed on the data set for all attributes, and the attribute with the highest gain:

$$A = \arg\max_{a \in SA} I(n,p) - E(a).$$

is selected to be the root of the tree. The data set is then split into v subsets $\{N_1, ..., N_v\}$, where the value of A for the objects in N_i is A_i, and for each subset, a corresponding node is created below the root. This process is repeated recursively for each node of the tree with the subset it contains until all the objects of the node are either positive or negative. For a training set of N instances each having M attributes, Quinlan (1986) showed that ID3's complexity to generate a decision tree is $O(NM)$.

5.4 Classification Using Linear Methods

5.4.1 Linear Classifiers

Decision trees are simple and efficient devices to design classifiers. Together with the information gain, they enabled us to induce optimal trees from a set of examples and to deal with symbolic values such as *sunny*, *hot*, and *high*.

Table 5.8. The frequency of *A* in the chapters of *Salammbô* in English and French. Letters have been normalized in uppercase and duplicate spaces removed

Chapter	**French**		**English**	
	# Characters	# *A*	# Characters	# *A*
Chapter 1	36,961	2,503	35,680	2,217
Chapter 2	43,621	2,992	42,514	2,761
Chapter 3	15,694	1,042	15,162	990
Chapter 4	36,231	2,487	35,298	2,274
Chapter 5	29,945	2,014	29,800	1,865
Chapter 6	40,588	2,805	40,255	2,606
Chapter 7	75,255	5,062	74,532	4,805
Chapter 8	37,709	2,643	37,464	2,396
Chapter 9	30,899	2,126	31,030	1,993
Chapter 10	25,486	1,784	24,843	1,627
Chapter 11	37,497	2,641	36,172	2,375
Chapter 12	40,398	2,766	39,552	2,560
Chapter 13	74,105	5,047	72,545	4,597
Chapter 14	76,725	5,312	75,352	4,871
Chapter 15	18,317	1,215	18,031	1,119
Total	619,431	42,439	608,230	39,056

Linear classifiers are another set of techniques that have the same purpose. As with decision trees, they produce a function splitting a set of objects into two or more classes. This time, however, the objects will be represented by a vector of numerical parameters. Such parameters are often called the **features** or the predictors. In the next sections, we examine linear classification methods in an n-dimensional space, where the dimension of the vector space is equal to the number of features used to characterize the objects.

5.4.2 Choosing a Data Set

To illustrate linear classification in a two-dimensional space, we will use *Salammbô* again in its original French version and in an English translation, and we will try to predict automatically the language of the version. As features, we will use the letter counts in each chapter: how many *A*, *B*, *C*, etc. The distribution of letters is different across both languages, for instance, *W* is quite frequent in English and rare in French. This makes it possible to use distribution models as an elementary method to identify the language of a text.

Although a more realistic language guesser would use all the letters of the alphabet, we will restrict it to *A*. We will count the total number of characters and the frequency of *A*s in each of the 15 chapters and try to derive a model from the data. Table 5.8 shows these counts in French and in English.

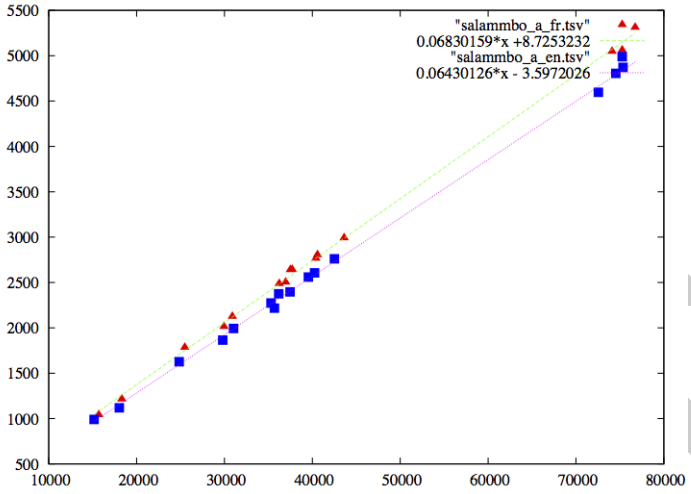

Fig. 5.11. Plot of the frequencies of *A*, *y*, versus the total character counts, *x*, in the 15 chapters of *Salammbô*. *Squares* correspond to the English version and *triangles* to the French original

5.5 Linear Regression

Before we try to discriminate between French and English, let us examine how we can model the distribution of the letters in one language.

Figure 5.11 shows the plot of data in Table 5.8, where each point represents the letter counts in one of the 15 chapters. The *x*-axis corresponds to the total count of letters in the chapter, and the *y*-axis, the count of *A*s. We can see from the figure that the points in both languages can be fitted quite precisely to two straight lines. This fitting process is called a linear regression, where a line equation is given by:

$$y = mx + b.$$

To determine the m and b coefficients, we will minimize a fitting error between the point distribution given by the set of q observations: $\{(x_i, y_i)\}_{i=1}^{q}$ and a perfect linear alignment given by the set $\{(x_i, f(x_i))\}_{i=1}^{q}$, where $f(x_i) = mx_i + b$. In our data set, we have 15 observations from each chapter in *Salammbô*, and hence $q = 15$.

5.5.1 Least Squares

The least squares method is probably the most common technique used to model the fitting error and estimate m and b. This error, or **loss function**, is defined as the sum of the squared errors (SSE) over all the q points (Legendre, 1805):

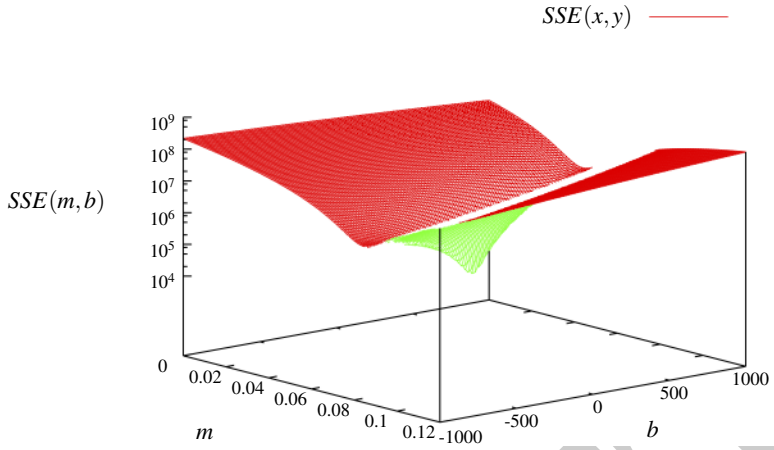

Fig. 5.12. Plot of $SSE(m,b)$ applied to the 15 chapters of the English version of *Salammbô*

$$\begin{aligned} SSE(m,b) &= \sum_{i=1}^{q} (y_i - f(x_i))^2, \\ &= \sum_{i=1}^{q} (y_i - (mx_i + b))^2. \end{aligned}$$

Ideally, all the points would be aligned and this sum would be zero. This is rarely the case in practice, and we fall back to an approximation that minimizes it.

Figure 5.5.1 shows the plot of $SSE(m,b)$ applied to the 15 chapters of the English version of *Salammbô*. Using a logarithmic scale, the surface shows a visible minimum somewhere between 0.6 and 0.8 for m and close to 0 for b. Let us now compute precisely these values.

We know from differential calculus that we reach the minimum of $SSE(m,b)$ when its partial derivatives over m and b are zero:

$$\begin{aligned} \frac{\partial SSE(m,b)}{\partial m} &= \sum_{i=1}^{q} \frac{\partial}{\partial m}(y_i - (mx_i + b))^2 = -2\sum_{i=1}^{q} x_i(y_i - (mx_i + b)) = 0. \\ \frac{\partial SSE(m,b)}{\partial b} &= \sum_{i=1}^{q} \frac{\partial}{\partial b}(y_i - (mx_i + b))^2 = -2\sum_{i=1}^{q} (y_i - (mx_i + b)) = 0. \end{aligned}$$

We obtain then:

$$m = \frac{\sum_{i=1}^{q} x_i y_i - q\overline{xy}}{\sum_{i=1}^{q} x_i^2 - q\overline{x}^2} \quad \text{and} \quad b = \overline{y} - m\overline{x},$$

with

$$\bar{x} = \frac{1}{q}\sum_{i=1}^{q} x_i \quad \text{and} \quad \bar{y} = \frac{1}{q}\sum_{i=1}^{q} y_i.$$

Using these formulas, we find the two regression lines for French and English:

$$\begin{aligned} &\text{French:} \quad y = 0.0683x + 8.7253 \\ &\text{English:} \quad y = 0.0643x - 3.5972 \end{aligned}$$

Least Absolute Deviation.

An alternative loss function to the least squares is to minimize the sum of the absolute errors (SAE) (Boscovich, 1770, Livre V, note):

$$SAE(m,b) = \sum_{i=1}^{q} |y_i - f(x_i)|.$$

The corresponding minimum value is called the least absolute deviation (LAD). Solving methods to find this minimum use linear programming. Their description falls outside the scope of this book.

Notations in an n-dimensional Space.

Up to now, we have formulated the regression problem with two parameters: the letter count and the count of *A*s. In most practical cases, we will have a much larger set. To describe algorithms applicable to any number of parameters, we need to extend our notation to a general n-dimensional space. Let us introduce it now.

In an n-dimensional space, it is probably easier to describe linear regression as a prediction technique: given input parameters in the form of a feature vector, predict the output value. In the *Salammbô* example, the input would be the number of letters in a chapter, and the output, the number of *A*s.

In a typical data set such as the one shown in Table 5.8, we have:

The input parameters: These parameters describe the observations we will use to predict an output. They are also called **feature vectors** or predictors, and we denote them $(1, x_1, x_2, ..., x_{n-1})$ or $\mathbf{x}$. The first parameter is set to 1 to make the computation easier. In the *Salammbô* example, this corresponds to the letter count in a chapter, for example: (1, 36961) in Chapter 1 in French. To denote the whole data set, we use a matrix, $\mathbf{X}$, where each row corresponds to an observation.

The output value: Each output, or response, represents the answer to a feature vector, and we denote it y, when we observe it, or $\hat{y}$, when we predict it using the regression line. In *Salammbô*, in French, the count of *A*s is $y = 2503$ in Chapter 1, and the predicted value using the regression line is $\hat{y} = 0.0683 \times 36961 + 8.7253 = 2533.22$. We use a vector, $\mathbf{y}$, to denote all the outputs in the data set.

The squared error: The squared error is the squared difference between the observed value and the prediction, $(y - \hat{y})^2$. In *Salammbô*, the squared error for Chapter 1 is $(2503 - 2533.22)^2 = 30.22^2 = 913.26$.

As we have seen, to compute the regression line, the least-squares method minimizes the sum of the squared errors for all the observations (here all the chapters). It is defined by its coefficients m and b in a two-dimensional space. In an n-dimensional space, we have:

The weight vector: The equivalent of a regression line when $n > 2$ is a hyperplane with a coefficient vector denoted $(w_0, w_1, w_2, ..., w_{n-1})$ or $\mathbf{w}$. These coefficients are usually called the weights. They correspond to (b, m) when $n = 2$. In *Salammbô*, the weight vector would be $(8.7253, 0.0683)$ for French and $(-3.5972, 0.0643)$ for English.

The intercept: This is the first weight w_0 of the weight vector. It corresponds to b when $n = 2$.

The hyperplane equation is given by the dot product of the weights by the feature variables. It is defined as:

$$y = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{n-1} w_i x_i,$$

where $\mathbf{w} = (w_0, w_1, w_2, ..., w_{n-1})$, $\mathbf{x} = (x_0, x_1, x_2, ..., x_{n-1})$, and $x_0 = 1$. Using a matrix notation, the predicted values, $\hat{\mathbf{y}}$, for all the observations in the data set, $\mathbf{X}$, are given by:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}.$$

For the French dataset, the complete matrix and vectors are:

$$\mathbf{X} = \begin{bmatrix} 1 & 36961 \\ 1 & 43621 \\ 1 & 15694 \\ 1 & 36231 \\ 1 & 29945 \\ 1 & 40588 \\ 1 & 75255 \\ 1 & 37709 \\ 1 & 30899 \\ 1 & 25486 \\ 1 & 37497 \\ 1 & 40398 \\ 1 & 74105 \\ 1 & 76725 \\ 1 & 18317 \end{bmatrix}; \mathbf{w} = \begin{bmatrix} 8.7253 \\ 0.0683 \end{bmatrix}; \hat{\mathbf{y}} = \begin{bmatrix} 2533.22 \\ 2988.11 \\ 1080.65 \\ 2483.36 \\ 2054.02 \\ 2780.95 \\ 5148.76 \\ 2584.31 \\ 2119.18 \\ 1749.46 \\ 2569.83 \\ 2767.97 \\ 5070.21 \\ 5249.16 \\ 1259.81 \end{bmatrix}; \mathbf{y} = \begin{bmatrix} 2503 \\ 2992 \\ 1042 \\ 2487 \\ 2014 \\ 2805 \\ 5062 \\ 2643 \\ 2126 \\ 1784 \\ 2641 \\ 2766 \\ 5047 \\ 5312 \\ 1215 \end{bmatrix}; \mathbf{se} = \begin{bmatrix} 913.26 \\ 15.14 \\ 1493.86 \\ 13.25 \\ 1601.31 \\ 578.40 \\ 7527.51 \\ 3444.53 \\ 46.57 \\ 1193.04 \\ 5065.18 \\ 38920 \\ 538.909 \\ 3948.29 \\ 2007.53 \end{bmatrix}.$$

5.5.2 The Gradient Descent

Using partial derivatives, we have been able to find an analytical solution to the regression line. We will now introduce the gradient descent, a generic optimization method that uses a series of successive approximations instead. We will apply this

technique to solve the least squares as well as the classification problems we will see in the next section.

The gradient descent (Cauchy, 1847) is a numerical method to find a global or local minimum of a function:

$$\begin{aligned} y &= f(x_0, x_1, x_2, ..., x_n), \\ &= f(\mathbf{x}), \end{aligned}$$

even when there is no analytical solution.

As we can see on Fig. 5.5.1, the sum of squared errors has a minimum. This a general property of the least squares, and the idea of the gradient descent is to derive successive approximations in the form of a sequence of points $(\mathbf{x}_p)$ to find it. At each iteration, the current point will move one step down to the minimum. For a function f, this means that we will have the inequalities:

$$f(\mathbf{x}_1) > f(\mathbf{x}_2) > ... > f(\mathbf{x}_k) > f(\mathbf{x}_{k+1}) > ... > min.$$

Now given a point $\mathbf{x}$, how can we find the next point of the iteration? The steps in the gradient descent are usually small and we can define the points in the neighborhood of $\mathbf{x}$ by $\mathbf{x}+\mathbf{v}$, where $\mathbf{v}$ is a vector of $\mathbb{R}^n$ and $||\mathbf{v}||$ is small. So the problem of gradient descent can be reformulated as: given $\mathbf{x}$, find $\mathbf{v}$ subject to $f(\mathbf{x}) > f(\mathbf{x}+\mathbf{v})$.

As $||\mathbf{v}||$ is small, we can approximate $f(\mathbf{x}+\mathbf{v})$ using a Taylor expansion limited to the first derivatives:

$$f(\mathbf{x}+\mathbf{v}) \approx f(\mathbf{x}) + \mathbf{v} \cdot \nabla f(\mathbf{x}),$$

where the gradient defined as:

$$\nabla f(x_0, x_1, x_2, ..., x_n) = (\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n})$$

is a direction vector corresponding to the steepest slope.

We obtain the steepest descent (respectively, ascent) when we choose $\mathbf{v}$ collinear to $\nabla f(\mathbf{x})$: $\mathbf{v} = -\alpha \nabla f(\mathbf{x})$ (respectively, $\mathbf{v} = \alpha \nabla f(\mathbf{x})$) with $\alpha > 0$. We have then:

$$f(\mathbf{x} - \alpha \nabla f(\mathbf{x})) \approx f(\mathbf{x}) - \alpha ||\nabla f(\mathbf{x})||^2,$$

and thus the inequality:

$$f(\mathbf{x}) > f(\mathbf{x} - \alpha \nabla f(\mathbf{x})).$$

This inequality enables us to write a recurrence relation between the steps:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)$$

and find a step sequence to the minimum, where α_k is a small positive number called the step size or learning rate. It can be constant over all the descent or change at each step. The convergence stops when $||\nabla f(\mathbf{x})||$ is less than a predefined threshold. This convergence is generally faster if the learning rate decreases over the iterations.

5.5.3 The Gradient Descent and Linear Regression

For a data set, *DS*, we find the minimum of the sum of squared errors and the coefficients of the regression equation through a walk down the surface using the recurrence relation above. Let us compute the gradient in a two-dimensional space first and then generalize it to multidimensional space.

In a Two-dimensional Space.

To make the generalization easier, let us rename the straight line coefficients (b,m) in $y = mx+b$ as (w_0,w_1). We want then to find the regression line:

$$\hat{y} = w_0 + w_1 x_1$$

given a data set *DS* of q examples: $DS = \{(1,x_1^j,y^j)|j : 1..q\}$, where the error is defined as:

$$\begin{aligned} SSE(w_0,w_1) &= \sum_{j=1}^{q} (y^j - \hat{y}^j)^2 \\ &= \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j))^2. \end{aligned}$$

The gradient of this two-dimensional equation $\nabla SSE(\mathbf{w})$ is:

$$\begin{aligned} \frac{\partial SSE(w_0,w_1)}{\partial w_0} &= -2\sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j)) \\ \frac{\partial SSE(w_0,w_1)}{\partial w_1} &= -2\sum_{j=1}^{q} x_1^j \times (y^j - (w_0 + w_1 x_1^j)). \end{aligned}$$

From this gradient, we can now compute the iteration step. With q examples and a learning rate of $\frac{\alpha}{2q}$, inversely proportional to the number of examples, we have:

$$\begin{aligned} w_0 &\leftarrow w_0 + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j)) \\ w_1 &\leftarrow w_1 + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} x_1^j \times (y^j - (w_0 + w_1 x_1^j)). \end{aligned}$$

In the iteration above, we compute the gradient as a sum over all the examples before we carry out one update of the weights. This technique is called the **batch gradient descent**. An alternate technique is to go through *DS* and compute an update with each example:

$$\begin{aligned} w_0 &\leftarrow w_0 + \alpha \cdot (y^j - (w_0 + w_1 x_1^j)) \\ w_1 &\leftarrow w_1 + \alpha \cdot x_1^j \cdot (y^j - (w_0 + w_1 x_1^j)). \end{aligned}$$

The examples are usually selected randomly from *DS*. This is called the **stochastic gradient descent** or **online learning**.

The duration of the descent is measured in **epochs**, where an epoch is the period corresponding to one iteration over the complete data set: the q examples.The stochastic variant often has a faster convergence.

N-dimensional Space.

In an n-dimensional space, we want to find the regression hyperplane:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n,$$

given a data set *DS* of q examples: $DS = \{(1, x_1^j, x_2^j, ..., x_n^j, y^j) | j : 1..q\}$, where the error is defined as:

$$\begin{aligned} SSE(w_0, w_1, ..., w_n) &= \sum_{j=1}^{q} (y^j - \hat{y}^j)^2 \\ &= \sum_{j=1}^{q} (y^j - (w_0 + w_1 x_1^j + w_2 x_2^j + ... + w_n x_n^j))^2. \end{aligned}$$

To simplify the computation of partial derivatives, we introduce the parameter $x_0^j = 1$ so that:

$$SSE(w_0, w_1, ..., w_n) = \sum_{j=1}^{q} (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + ... + w_n x_n^j))^2.$$

The gradient of *SSE* is:

$$\frac{\partial SSE}{\partial w_i} = -2 \sum_{j=1}^{q} x_i^j \times (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + ... + w_n x_n^j)).$$

In the batch version, the iteration step considers all the examples in *DS*:

$$w_i \leftarrow w_i + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} x_i^j \cdot (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + ... + w_n x_n^j)).$$

or using a matrix notation:

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha}{q} \cdot \mathbf{X}^\intercal (\mathbf{y} - \hat{\mathbf{y}}),$$

where $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$.

In the stochastic version, we carry out the updates using one example at a time. For the jth example corresponding to $\mathbf{x}^j$ and y^j, it results in the update:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot (y^j - \hat{y}^j) \cdot \mathbf{x}^j,$$

where $\hat{y}^j = \mathbf{x}^j \cdot \mathbf{w}$.

5.6 Regularization

5.6.1 The Analytical Solution Again

In Sect. 5.5.1, we saw that linear regression in a two-dimensional space had an analytical solution. This can be generalized for any dimension.

Let $\mathbf{X} = (\mathbf{x}^1; \mathbf{x}^2; ...; \mathbf{x}^q)$ be our predictors and $\mathbf{y}$, the output. Following the formulation in Sect. 5.5.1, linear regression consists in finding the weight vector $\mathbf{w}$ that minimizes the sum of squared errors between $\mathbf{Xw}$ and $\mathbf{y}$. Ideally, we would have this equality:

$$\mathbf{Xw} = \mathbf{y}.$$

To solve this equation, we could think of inverting $\mathbf{X}$ and have:

$$\mathbf{w} = \mathbf{X}^{-1}\mathbf{y}.$$

Unfortunately, this simple operation is not possible in general as $\mathbf{X}$ is not a square matrix and hence not invertible. Nonetheless, Fredholm (1903) and then Moore (1920) showed that we can define a **pseudoinverse** of $\mathbf{X}$ in the form of:

$$(\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal$$

that solves the least square problem:

$$\mathbf{w} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}.$$

5.6.2 Inverting $\mathbf{X}^\intercal\mathbf{X}$

At this point, we may wonder if we can always derive a pseudoinverse. To do this, we need to invert $\mathbf{X}^\intercal\mathbf{X}$, which is possible if $\mathbf{X}$ has linearly independent columns. In real data sets, it is frequently the case that we have highly correlated columns or even duplicate ones. This means that $\mathbf{X}^\intercal\mathbf{X}$ will be singular, and hence not invertible.

A way to make $\mathbf{X}^\intercal\mathbf{X}$ invertible is to add it a scalar matrix $\lambda\mathbf{I}$, where λ is small (Hoerl, 1962):

$$\mathbf{w} = (\mathbf{X}^\intercal\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\intercal\mathbf{y}$$

This operation is called a **regularization** and is equivalent to adding the term $\lambda||\mathbf{w}||^2$ to the sum of squared errors (*SSE*). It is also used in classification.

5.6.3 Regularization

Correlated features result in large values of $\mathbf{w}$ that regularization tries to penalize. In most practical regression and classification cases, we replace *Loss*, the sum of squared errors in regression, with a *Cost*:

$$Cost(\mathbf{w}) = Loss(\mathbf{w}) + \lambda L_q(\mathbf{w}),$$

where

$$L_q = \sum_{i=1}^{n} |w_i|^q.$$

The most frequent regularizations are (ridge regression):

$$L_2 = \sum_{i=1}^{n} w_i^2,$$

and (LASSO regression):

$$L_1 = \sum_{i=1}^{n} |w_i|.$$

In practice, w_0 is often part of the regularization.

5.7 Linear Classification

5.7.1 An Example

We will now use the data set in Table 5.8 to describe classification techniques that split the texts into French or English. If we examine it closely, Fig. 5.11 shows that we can draw a straight line between the two regression lines to separate the two classes. This is the idea of linear classification. From a data representation in a Euclidian space, classification will consist in finding a line:

$$w_0 + w_1 x + w_2 y = 0$$

separating the plane into two half-planes defined by the inequalities:

$$w_0 + w_1 x + w_2 y > 0$$

and

$$w_0 + w_1 x + w_2 y < 0.$$

These inequalities mean that the points belonging to one class of the data set are on one side of the separating line and the others are on the other side.

In Table 5.8 and Fig. 5.11, the chapters in French have a steeper slope that the corresponding ones in English. The points representing the French chapters will then be above the separating line. Let us write the inequalities that reflect this and set w_2 to 1 to normalize them. The line we are looking for will have the property:

$$\begin{array}{ll} y_i > w_0 + w_1 x_i & \text{for the set of points: } \{(x_i, y_i) | (x_i, y_i) \in \text{French}\} \text{ and} \\ y_i < w_0 + w_1 x_i & \text{for the set of points: } \{(x_i, y_i) | (x_i, y_i) \in \text{English}\}, \end{array}$$

where x is the total count of letters in a chapter and y, the count of *A*s. In total, we will have 30 inequalities, 15 for French and 15 for English shown in Table 5.9. Any weight vector $\mathbf{w} = (w_0, w_1)$ that satisfies all of them will define a classifier correctly separating the chapters into two classes: French or English.

Table 5.9. Inequalities derived from Table 5.8 for the 15 chapters in *Salammbô* in French and English

Chapter	French	English
1	$2503 > w_0 + 36961w_1$	$2217 < w_0 + 35680w_1$
2	$2992 > w_0 + 43621w_1$	$2761 < w_0 + 42514w_1$
3	$1042 > w_0 + 15694w_1$	$990 < w_0 + 15162w_1$
4	$2487 > w_0 + 36231w_1$	$2274 < w_0 + 35298w_1$
5	$2014 > w_0 + 29945w_1$	$1865 < w_0 + 29800w_1$
6	$2805 > w_0 + 40588w_1$	$2606 < w_0 + 40255w_1$
7	$5062 > w_0 + 75255w_1$	$4805 < w_0 + 74532w_1$
8	$2643 > w_0 + 37709w_1$	$2396 < w_0 + 37464w_1$
9	$2126 > w_0 + 30899w_1$	$1993 < w_0 + 31030w_1$
10	$1784 > w_0 + 25486w_1$	$1627 < w_0 + 24843w_1$
11	$2641 > w_0 + 37497w_1$	$2375 < w_0 + 36172w_1$
12	$2766 > w_0 + 40398w_1$	$2560 < w_0 + 39552w_1$
13	$5047 > w_0 + 74105w_1$	$4597 < w_0 + 72545w_1$
14	$5312 > w_0 + 76725w_1$	$4871 < w_0 + 75352w_1$
15	$1215 > w_0 + 18317w_1$	$1119 < w_0 + 18031w_1$

Let us represent graphically the inequalities in Table 5.9 and solve the system in the two-dimensional space defined by w_0 and w_1. Figure 5.13 shows a plot with the two first chapters, where w_1 is the abscissa and w_0, the ordinate. Each inequality defines a half-plane that restricts the set of possible weight values. The four inequalities delimit the solution region in white, where the two upper lines are constraints applied by the two chapters in French and the two below by their English translations.

Figure 5.14 shows the plot for all the chapters. The remaining inequalities shrink even more the polygonal region of possible values. The point coordinates (w_1, w_0) in this region, as, for example, $(0.066, 0)$ or $(0.067, -20)$, will satisfy all the inequalities and correctly separate the 30 observations into two classes: 15 chapters in French and 15 in English.

5.7.2 Classification in an *N*-dimensional Space

In the example above, we used a set of two-dimensional points, (x_i, y_i) to represent our observations. This process can be generalized to vectors in a space of dimension n. The separator will then be a hyperplane of dimension $n-1$. In a space of dimension 2, a hyperplane is a line; in dimension 3, a hyperplane is a plane of dimension 2, etc. In an n-dimensional space, the inequalities defining the two classes will be:

$$w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n > 0$$

and

$$w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n < 0,$$

where each observation is described by a feature vector $\mathbf{x}$.

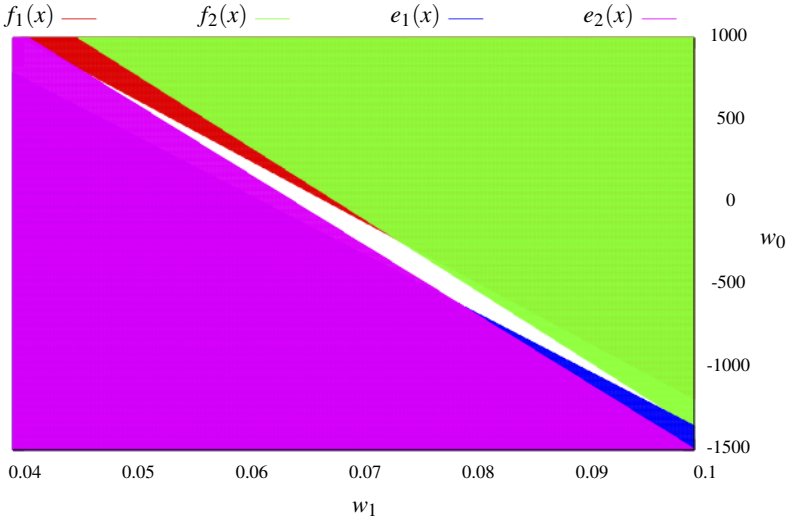

Fig. 5.13. A graphical representation of the inequality system restricted to the two first chapters in French, f_1 and f_2, and in English, e_1 and e_2. We can use any point coordinates in the *white region* as parameters of the line to separate these two chapters

The sums in the inequalities correspond to the dot product of the weight vector, **w**, by the the feature vector, **x**, defined as

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{n} w_i x_i,$$

where $\mathbf{w} = (w_0, w_1, w_2, ..., w_n)$, $\mathbf{x} = (x_0, x_1, x_2, ..., x_n)$, and $x_0 = 1$.

The purpose of the classification algorithms is to find lines or hyperplanes separating most accurately a set of data represented by numerical vectors into two classes. As with the decision trees, these separators will be approximated from training sets and evaluated on distinct test sets. We will review the vocabulary used with machine learning methods in more detail in Sect. 7.3.2.

5.7.3 Linear Separability

It is not always the case that a line can perfectly separate the two classes of a data set. Let us return to our data set in Table 5.8 and restrict ourselves to the three shortest chapters: the 3rd, 10th, and 15th. Figure 5.15, left, shows the plot of these three chapters from the counts collected in the actual texts. A thin line can divide the chapters into two classes. Now let us imagine that in another data set, Chapter 10 in French has 18,317 letters and 1,115 *A*s instead of 18,317 and 1,215, respectively. Figure 5.15,

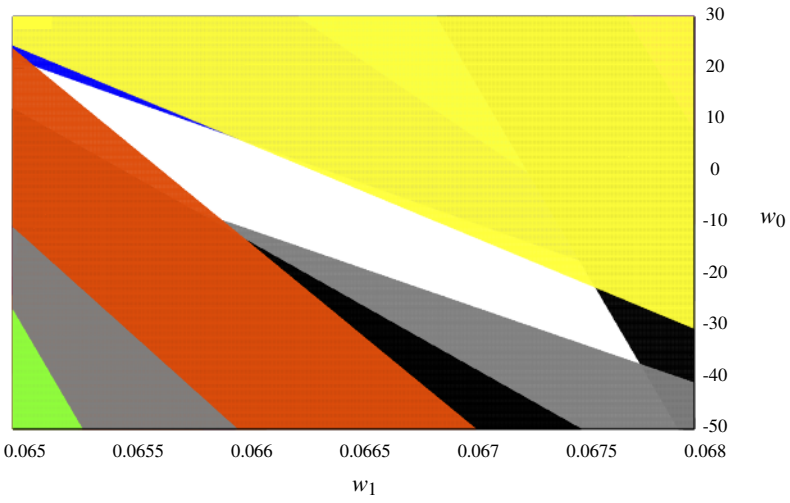

Fig. 5.14. A graphical representation of the inequality system with all the chapters. The point coordinates in the *white polygonal region* correspond to weights vectors (w_1, w_0) defining a separating line for all the chapters

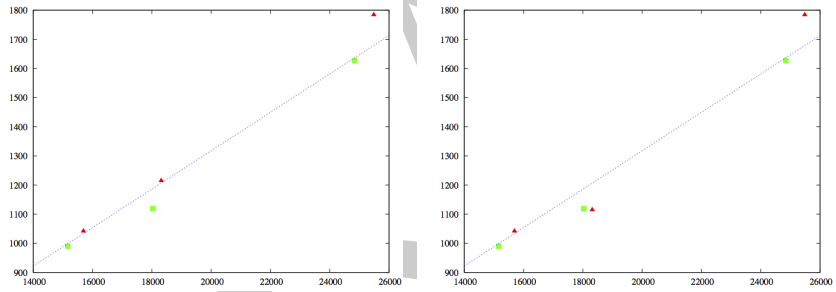

Fig. 5.15. *Left part*: A thin line can separate the three chapters into French and English text. The two classes are linearly separable. *Right part*: We cannot draw a line between the two classes. They are not linearly separable

right, shows this plot. This time, no line can pass between the two classes, and the data set is said to be not linearly separable.

Although we cannot draw a line that divides the two classes, there are workarounds to cope with not linearly separable data that we will explain in the next section.

5.7.4 Classification vs. Regression

Regression and classification use a similar formalism, and at this point, it is important to understand their differences. Given an input, regression computes a continuous numerical output. For instance, regression will enable us to compute the number of *A*s occurring in a text in French from the total number of characters. Having 75,255 characters in Chapter 7, the regression line will predict 5,149 occurrences of *A*s (there are 5,062 in reality).

The output of a classification is a finite set of values. When there are two values, we have a binary classification. Given the number of characters and the number of *A*s in a text, classification will predict the language: French or English. For instance, having the pair (75255, 5062), the classifier will predict French.

This means that given a data set, the dimensions of the feature space will be different. Regression predicts the value of one of the features given the value of $n-1$ features. Classification predicts the class given the values of n features. Compared to regression in our example, the dimension of the vector space used for the classification is $n+1$: the n features and the class.

In the next sections, we will examine three categories of linear classifiers from among the most popular and efficient ones: perceptrons, logistic regression, and support vector machines. For the sake of simplicity, we will restrict our presentation to a binary classification with two classes. However, linear classifiers can generalize to handle a multinomial classification, *i.e.* three classes or more, which is the most frequent case in practice. This generalization is outside the scope of this book; see Sect. 5.15 for further references on this topic.

5.8 Perceptron

Given a data set like the one in Table 5.8, where each object is characterized by the feature vector $\mathbf{x}$ and a class, P or N, the perceptron algorithm (Rosenblatt, 1958) is a simple method to find a hyperplane splitting the space into positive and negative half-spaces separating the objects. The perceptron uses a sort of gradient descent to iteratively adjust weights $(w_0, w_1, w_2, ..., w_n)$ representing the hyperplane until all the objects belonging to P have the property $\mathbf{w} \cdot \mathbf{x} \geq 0$, while those belonging to N have a negative dot product.

5.8.1 The Heaviside Function

As we represent the examples using numerical vectors, it is more convenient in the computations to associate the negative and positive classes, N and P, to a discrete set of two numerical values: $\{0,1\}$. To carry this out, we pass the result of the dot product to the Heaviside step function (a variant of the *signum* function):

$$H(\mathbf{w} \cdot \mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Using the Heaviside function H, we can reformulate classification. Given a data set : $DS = \{(1, x_1^j, x_2^j, ..., x_n^j, y^j) | j : 1..q\}$ of q examples, where $y^j \in \{0,1\}$, we have:

$$\begin{aligned}\hat{y}(\mathbf{x}^j) &= H(\mathbf{w} \cdot \mathbf{x}^j), \\ &= H(w_0 + w_1 x_1^j + w_2 x_2^j + ... + w_n x_n^j).\end{aligned}$$

We use $x_0^j = 1$ to simplify the equations, and the range of y, $\{0,1\}$, corresponds to the classes {English, French} in Table 5.8.

5.8.2 The Iteration

Let us denote $\mathbf{w}_k$ the weight vector at step k, and $w_{i(k)}$, the value of its weight coordinate w_i. The perceptron algorithm starts the iteration with a weight vector $\mathbf{w}_0$ chosen randomly or set to $\mathbf{0}$ and then applies the dot product $\mathbf{w}_k \cdot \mathbf{x}^j$ one object at a time for all the members of the data set, $j : 1..q$:

- If the object is correctly classified, the perceptron algorithm keeps the weights unchanged;
- If the object is misclassified, the algorithm attempts to correct the error by adjusting $\mathbf{w}_k$ using a gradient descent:

$$w_{i(k+1)} \leftarrow w_{i(k)} + \alpha \cdot (y^j - \hat{y}^j) \cdot x_i^j.$$

until all the objects are correctly classified. With a vector notation, this corresponds to:

$$\mathbf{w}_{(k+1)} \leftarrow \mathbf{w}_{(k)} + \alpha \cdot (y^j - \hat{y}^j) \cdot \mathbf{x}^j.$$

For a misclassified object, we have $y^j - \hat{y}^j$ equals to either $1-0$ or $0-1$. The update value is then is $\alpha \cdot x_i$ or $-\alpha \cdot x_i$, where α is the learning rate. For an object that is correctly classified, we have $y^j - \hat{y}^j = 0$, corresponding to either $0-0$ or $1-1$, and there is no weight update. The learning rate is generally set to 1 as a division of the weight vector by a constant does not affect the update rule.

The perceptron also has a batch version defined by the update rule:

$$\mathbf{w}_{(k+1)} \leftarrow \mathbf{w}_{(k)} + \frac{\alpha}{q} \mathbf{X}^\intercal \cdot (\mathbf{y} - \hat{\mathbf{y}}).$$

5.8.3 The Two-Dimensional Case

Let us spell out the update rules in a two-dimensional space. We have the feature vectors and weight vectors defined as: $\mathbf{x} = (1, x_1, x_2)$ and $\mathbf{w} = (w_0, w_1, w_2)$. With the stochastic gradient descent, we carry out the updates using the relations:

$$\begin{aligned}w_0 &\leftarrow w_0 + y^j - \hat{y}^j \\ w_1 &\leftarrow w_1 + (y^j - \hat{y}^j) \cdot x_1^j \\ w_2 &\leftarrow w_2 + (y^j - \hat{y}^j) \cdot x_2^j,\end{aligned}$$

where $y^j - \hat{y}^j$ is either, 0, -1, or 1.

5.8.4 Stop Conditions

To find a hyperplane, the objects (i.e., the points) must be separable. This is rarely the case in practice, and we often need to refine the stop conditions. We will stop the learning procedure when the number of misclassified examples is below a certain threshold or we have exceeded a fixed number of iterations.

The perceptron will converge faster if, for each iteration, we select the objects randomly from the data set.

5.9 Support Vector Machines

When a data set is linearly separable, the perceptron algorithm finds a separating hyperplane with a weight vector corresponding to a point in the solution region. Referring back to Fig. 5.14, it can be any point in the white region.

Support vector machines (Boser et al., 1992) are another type of linear classifiers that aim at finding a unique solution in the form of an optimal hyperplane. This optimal hyperplane is defined as the one that maximizes the margins between the two classes. It will be positioned at equal distance between the closest points of each class and will create the largest possible corridor with no points from either classes inside. These closest points are on the border of the corridor and are called the support vectors. In this section, we introduce the mathematical concepts behind support vector machines.

5.9.1 Maximizing the Margin

We know from geometry and vector analysis that the distance from a point $\mathbf{x}^j = (x_1^j, x_2^j, ..., x_n^j)$ to a hyperplane Hyp defined by the equation:

$$w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n = 0$$

is given by the formula:

$$\begin{aligned} d(\mathbf{x}^j, Hyp) &= \frac{|w_0 + w_1 x_1^j + w_2 x_2^j + ... + w_n x_n^j|}{\sqrt{w_1^2 + w_2^2 + ... + w_n^2}} \\ &= \frac{|w_0 + \mathbf{w} \cdot \mathbf{x}^j|}{||\mathbf{w}||}, \end{aligned}$$

where $\mathbf{w}$ is the weight vector $(w_1, w_2, ...w_n)$.

The optimal hyperplane is the one that maximizes this distance for all the points in the data set, $DS = \{(x_1^j, x_2^j, ..., x_n^j, y^j) | j : 1..q\}$. It is easier to associate the negative and positive classes, N and P, to the two numerical values: $\{-1, 1\}$ instead of $\{0, 1\}$ as in the perceptron. We can then remove the absolute value and fit the weight vector so that it maximizes the margin M:

$$\begin{aligned} &\max_{w_0,\mathbf{w}} \ M \\ &\text{subject to } \ y^j \cdot \frac{w_0 + \mathbf{x}^j \cdot \mathbf{w}}{||\mathbf{w}||} \geqslant M, j : 1..q, \end{aligned}$$

where $y^j \in \{-1, 1\}$.

The weight vector $(w_0, \mathbf{w})$ is defined within a constant factor, and we can set $||\mathbf{w}||$ so that $M = \frac{1}{||\mathbf{w}||}$. Using this scaling operation, maximizing M is equivalent to minimizing the norm $||\mathbf{w}||$. We have then

$$\begin{aligned} &\min_{w_0,\mathbf{w}} \ ||\mathbf{w}|| \\ &\text{subject to } \ y^j(w_0 + \mathbf{x}^j \cdot \mathbf{w}) \geqslant 1, j : 1..q. \end{aligned}$$

5.9.2 Lagrange Multipliers

The margin maximization can be recast using a Lagrangian (a Lagrange function) (Boser et al., 1992):

$$\begin{aligned} &L(\mathbf{w}, w_0, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{j=1}^{q} \alpha_j (y^j(w_0 + \mathbf{x}^j \cdot \mathbf{w}) - 1), \\ &\text{subject to } \alpha_j \geqslant 0, j : 1..q, \end{aligned}$$

and is then equivalent to finding a minimum of $L(\mathbf{w}, w_0, \alpha)$ with respect to $\mathbf{w}$ and a maximum with respect to α. The α are called Lagrange multipliers.

The maximal margin is reached when the partial derivatives with respect to $\mathbf{w}$ and w_0 are 0. Computing the derivatives, we find:

$$\begin{cases} \sum_{j=1}^{q} \alpha_j y^j x_i^j & = w_i \\ \sum_{j=1}^{q} \alpha_j y^j & = 0 \end{cases}$$

We plug these values back into the Lagrangian, and we obtain:

$$L(\alpha) = \sum_{j=1}^{q} \alpha_j - \frac{1}{2} \sum_{j=1}^{q} \sum_{k=1}^{q} \alpha_j \alpha_k y^j y^k \mathbf{x}^j \mathbf{x}^k,$$

that we maximize with respect to α.

We can find a solution to this optimization problem using quadratic programming techniques. Their description is beyond the scope of this book, however. There are many toolkits that we can use to solve practical problems. They include the LIBSVM (Chang and Lin, 2011) and LIBLINEAR (Fan et al., 2008) toolkits.

Applying LIBLINEAR to the data in Table 5.8, we find a hyperplane equation separating the two classes so that $||\mathbf{w}|| \cdot M = 1$:

$$-0.006090937 + 0.008155714x - 0.123790484y = 0$$

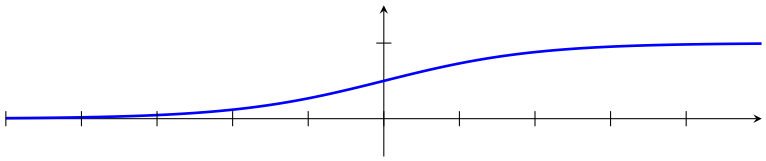

Fig. 5.16. The logistic curve: $f(x) = \dfrac{1}{1+e^{-x}}$

and when normalizing the coefficients with respect to y we have:

$$0.049203592 - 0.065883207x + y = 0$$

This corresponds to unique point $(w_0, w_1) = (-0.049203592, 0.065883207)$ in the weight space in Fig. 5.14.

Support vector machines can also handle not linearly separable examples using kernels or through the soft margin method. See the original papers by Boser et al. (1992) and Cortes and Vapnik (1995) for a presentation.

5.10 Logistic Regression

In their elementary formulation, the perceptron and support vector machines use hyperplanes as absolute, unmitigated boundaries between the classes. In many data sets, however, there are no such clear-cut thresholds to separate the points. Figure 5.15 is an example of this that shows regions where the nonlinearly separable classes have points with overlapping feature values.

Logistic regression is an attempt to define a smoother transition between the classes. Instead of a rigid boundary in the form of a step function, logistic regression uses the logistic curve (Verhulst, 1838, 1845) to model the probability of a point **x** (an observation) to belong to a class. Figure 5.16 shows this curve, whose equation is given by:

$$f(x) = \frac{1}{1+e^{-x}}.$$

The logistic curve is also called a **sigmoid**.

Logistic regression was first introduced by Berkson (1944) in an attempt to model the percentage of individuals killed by the intake of a lethal drug. Berkson observed that the higher the dosage of the drug, the higher the mortality, but as some individuals are more resilient than others, there was no threshold value under which all the individuals would have survived and above which all would have died. Intuitively, this fits very well the shape of the logistic curve in Fig. 5.16, where the mortality rate is close to 0 for lower values of x (the drug dosage), then increases, and reaches a mortality rate of 1 for higher values of x.

Berkson used one feature, the dosage x, to estimate the mortality rate, and he derived the probability model:

$$P(y=1|x) = \frac{1}{1+e^{-w_0-w_1 x}},$$

where y denotes the class, either survival or death, with the respective labels 0 and 1, and (w_0, w_1) are weight coefficients that are fit using the maximum likelihood method.

Using this assumption, we can write a general probability model for feature vectors $\mathbf{x}$ of any dimension:

$$P(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}},$$

where $\mathbf{w}$ is a weight vector.

As we have two classes and the sum of their probabilities is 1, we have:

$$P(y=0|\mathbf{x}) = \frac{e^{-\mathbf{w}\cdot\mathbf{x}}}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}.$$

These probabilities are extremely useful in practice.

The logit transformation corresponding to the logarithm of the odds ratio:

$$\ln\frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} = \ln\frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})} = \mathbf{w}\cdot\mathbf{x}$$

is also frequently used to fit the data to a straight line or a hyperplane.

5.10.1 Fitting the Weight Vector

To build a functional classifier, we need now to fit the weight vector $\mathbf{w}$; the maximum likelihood is a classical way to do this. Given a data set, $DS = \{(1, x_1^j, x_2^j, ..., x_n^j, y^j) | j : 1..q\}$, containing a partition in two classes, P ($y=1$) and N ($y=0$), and a weight vector $\mathbf{w}$, the likelihood to have the classification observed in this data set is:

$$L(\mathbf{w}) = \prod_{\mathbf{x}^j \in P} P(y^j = 1|\mathbf{x}^j) \times \prod_{\mathbf{x}^j \in N} P(y^j = 0|\mathbf{x}^j),$$

$$= \prod_{\mathbf{x}^j \in P} P(y^j = 1|\mathbf{x}^j) \times \prod_{\mathbf{x}^j \in N} (1 - P(y^j = 1|\mathbf{x}^j)).$$

We can rewrite the product using y^j as powers of the probabilities as $y^j = 0$, when $\mathbf{x}^j \in N$ and $y^j = 1$, when $\mathbf{x}^j \in P$:

$$L(\mathbf{w}) = \prod_{\mathbf{x}^j \in P} P(y^j = 1|\mathbf{x}^j)^{y^j} \times \prod_{\mathbf{x}^j \in N} (1 - P(y^j = 1|\mathbf{x}^j))^{1-y^j},$$

$$= \prod_{(\mathbf{x}^j, y^j) \in DS} P(y^j = 1|\mathbf{x}^j)^{y^j} \times (1 - P(y^j = 1|\mathbf{x}^j))^{1-y^j}.$$

Maximizing the Likelihood.

We fit $\mathbf{w}$, and train a model by maximizing the likelihood of the observed classification:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \prod_{\mathbf{x}^j \in DS} P(y^j = 1|\mathbf{x}^j)^{y^j} \times (1 - P(y^j = 1|\mathbf{x}^j))^{1-y^j}.$$

To maximize this term, it is more convenient to work with sums rather than with products, and we take the logarithm of it, the **log-likelihood**:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_{(\mathbf{x}^j, y^j) \in DS} y^j \ln P(y^j = 1|\mathbf{x}^j) + (1 - y^j) \ln(1 - P(y^j = 1|\mathbf{x}^j)).$$

Using the logistic curves to express the probabilities, we have:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_{(\mathbf{x}^j, y^j) \in DS} y^j \ln \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}^j}} + (1 - y^j) \ln \frac{e^{-\mathbf{w} \cdot \mathbf{x}^j}}{1 + e^{-\mathbf{w} \cdot \mathbf{x}^j}}.$$

In contrast to linear regression that uses least mean squares, here we fit a logistic curve so that it maximizes the likelihood of the classification – partition – observed in the training set.

Alternatively, we could minimize the negative log-likelihood. We then call the corresponding function the **log-loss** or **cross entropy**. However, differently to Sect. 5.2.3, most machine learning practitioners use the natural logarithm this time, and compute the mean:

$$-\frac{1}{q} \sum_{j=1}^{q} (y^j \ln \hat{y}^j + (1 - y^j) \ln(1 - \hat{y}^j)).$$

5.10.2 The Gradient Ascent

We can use the gradient ascent to compute this maximum. This method is analogous to the gradient descent that we reviewed in Sect. 5.5.2; we move upward instead. A Taylor expansion of the log-likelihood gives us: $\ell(\mathbf{w} + \mathbf{v}) = \ell(\mathbf{w}) + \mathbf{v} \cdot \nabla \ell(\mathbf{w}) + \ldots$ When $\mathbf{w}$ is collinear with the gradient, we have:

$$\ell(\mathbf{w} + \alpha \nabla \ell(\mathbf{w})) \approx \ell(\mathbf{w}) + \alpha ||\nabla \ell(\mathbf{w})||^2.$$

The inequality:

$$\ell(\mathbf{w}) < \ell(\mathbf{w} + \alpha \nabla \ell(\mathbf{w}))$$

enables us to find a sequence of increasing values of the log-likekihood. We use the iteration:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \nabla \ell(\mathbf{w}_k)$$

to carry this out until we reach a maximum.

Computing the Gradient.

We compute the partial derivatives of the log-likelihood to find the gradient:

$$\begin{aligned}
\frac{\partial \ell(\mathbf{w})}{\partial w_i} &= \sum_{(\mathbf{x}^j, y^j) \in DS} y^j (1 + e^{-\mathbf{w}.\mathbf{x}^j}) \frac{x_i^j e^{-\mathbf{w}.\mathbf{x}^j}}{(1 + e^{-\mathbf{w}\cdot\mathbf{x}^j})^2} + \\
&\quad (1 - y^j) \frac{1 + e^{-\mathbf{w}.\mathbf{x}^j}}{e^{-\mathbf{w}.\mathbf{x}^j}} \cdot \frac{-x_i^j \cdot e^{-\mathbf{w}\cdot\mathbf{x}^j}(1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}) + x_i^j \cdot e^{-\mathbf{w}.\mathbf{x}^j} \cdot e^{-\mathbf{w}.\mathbf{x}^j}}{(1 + e^{-\mathbf{w}\cdot\mathbf{x}^j})^2}, \\
&= \sum_{(\mathbf{x}^j, y^j) \in DS} y^j \frac{x_i^j e^{-\mathbf{w}.\mathbf{x}^j}}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}} + (1 - y^j) \cdot \frac{-x_i^j \cdot (1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}) + x_i^j \cdot e^{-\mathbf{w}.\mathbf{x}^j}}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}}, \\
&= \sum_{(\mathbf{x}^j, y^j) \in DS} x_i^j \cdot \frac{y^j \cdot (1 + e^{-\mathbf{w}.\mathbf{x}^j}) - 1}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}}, \\
&= \sum_{(\mathbf{x}^j, y^j) \in DS} x_i^j \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}}).
\end{aligned}$$

Weight Updates.

Using the gradient values, we can now compute the weight updates at each step of the iteration. As with linear regression, we can use a stochastic or a batch method. For $DS = \{(1, x_1^j, x_2^j, ..., x_n^j, y^j) | j : 1..q\}$, the updates of $\mathbf{w} = (w_0, w_1, ..., w_n)$ are:

- With the stochastic gradient ascent:

$$w_{i(k+1)} \leftarrow w_{i(k)} + \alpha \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}^j}}) \cdot x_i^j;$$

  or for the whole $\mathbf{w}$ vector:

$$\mathbf{w}_{(k+1)} \leftarrow \mathbf{w}_{(k)} + \alpha \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}^j}}) \cdot \mathbf{x}^j;$$

- With the batch gradient ascent:

$$w_{i(k+1)} \leftarrow w_{i(k)} + \frac{\alpha}{q} \cdot \sum_{j=1}^{q} x_i^j \cdot (y^j - \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}^j}}).$$

  or using a matrix notation:

$$\mathbf{w}_{(k+1)} \leftarrow \mathbf{w}_{(k)} + \frac{\alpha}{q} \mathbf{X}^\intercal \cdot (\mathbf{y} - \hat{\mathbf{y}}),$$

  where $\hat{y}^j = \frac{1}{1 + e^{-\mathbf{w}\cdot\mathbf{x}^j}}$.

As with the gradient descent, the convergence stops when $||\nabla \ell(\mathbf{w})||$ is less than a predefined threshold.

Table 5.10. A representation of the symbolic values in Table 5.7 as numerical vectors

Object	Attributes										Class
	Outlook			Temperature			Humidity		Windy		
	Sunny	Overcast	Rain	Hot	Mild	Cool	High	Normal	True	False	
1	1	0	0	1	0	0	1	0	0	1	*N*
2	1	0	0	1	0	0	1	0	1	0	*N*
3	0	1	0	1	0	0	1	0	0	1	*P*
4	0	0	1	0	1	0	1	0	0	1	*P*
5	0	0	1	0	0	1	0	1	0	1	*P*
6	0	0	1	0	0	1	0	1	1	0	*N*
7	0	1	0	0	0	1	0	1	1	0	*P*
8	1	0	0	0	1	0	1	0	0	1	*N*
9	1	0	0	0	0	1	0	1	0	1	*P*
10	0	0	1	0	1	0	0	1	0	1	*P*
11	1	0	0	0	1	0	0	1	1	0	*P*
12	0	1	0	0	1	0	1	0	1	0	*P*
13	0	1	0	1	0	0	0	1	0	1	*P*
14	0	0	1	0	1	0	1	0	1	0	*N*

5.11 Encoding Symbolic Values as Numerical Features

Along with this overview of numerical classification methods, a practical question comes to mind: how can we apply them to symbolic – or nominal – attributes like the ones in Table 5.7?

The answer is that we need to convert the symbolic attributes into numerical vectors before we can use the linear classifiers. The classical way to do this is to represent each attribute domain – the set of the allowed or observed values of an attribute – as a vector of binary digits. Let us exemplify this with the *Outlook* attribute in Table 5.7:

- *Outlook* has three possible values: $\{sunny, overcast, rain\}$. Its numerical representation is then a three-dimensional vector, (x_1, x_2, x_3), whose axes are tied respectively to *sunny*, *overcast*, and *rain*.
- To reflect the value of the attribute, we set the corresponding coordinate to 1 and the others to 0. Using the examples in Table 5.7, the name–value pair $[Outlook = sunny]$ will be encoded as $(1,0,0)$, $[Outlook = overcast]$ as $(0,1,0)$, and $[Outlook = rain]$ as $(0,0,1)$.

For a given attribute, the dimension of the vector will then be defined by the number of its possible values, and each vector coordinate will be tied to one of the possible values of the attribute.

So far, we have one vector for each attribute. To represent a complete object, we will finally concatenate all these vectors into a larger one characterizing this object. Table 5.10 shows the complete conversion of the data set using vectors of binary values. This type of encoding is also called **one-hot encoding**.

If an attribute has from the beginning a numerical value, it does not need to be converted. It is, however, a common practice to scale it so that the observed values in the training set range from 0 to 1 or from –1 to +1.

5.12 scikit-learn: A Machine–Learning Library

scikit-learn (Pedregosa et al., 2011) is a popular and comprehensive machine–learning library. It provides with a large set of supervised and unsupervised algorithms that we can use with Python. The `sklearn.linear_model` module, for instance, includes the perceptron and logistic regression, while `sklearn.svm` provides support-vector machine algorithms.

The classifiers use two main functions: `fit()` to train a model and `predict()` to predict a class. In the scikit-learn documentation, the functions adopt a notation, where:

- **x** denotes a feature vector (the predictors) describing one observation and **X**, a feature matrix representing the dataset;
- *y* denotes the class (or response or target) of one observation and **y**, the class vector for the whole dataset.

Both **X** and **y** must be in the numpy array format, where numpy is the numerical computation library on which scikit-learn is built. In the next sections, we will see how to use this format.

5.12.1 Numerical Data

In this section, we will learn how to load a dataset, fit a model, and predict classes with the scikit-learn functions. All the feature values in the dataset in Table 5.8 are numeric and creating **X** and **y** in a numpy array format is straightforward. We just need to decide on a convention on how to represent the classes: We assign 0 to English and 1 to French. We create the arrays with `np.array()` and the lists of values as arguments:

```
import numpy as np

y = np.array(
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

X = np.array(
    [[35680, 2217], [42514, 2761], [15162, 990], [35298, 2274],
     [29800, 1865], [40255, 2606], [74532, 4805], [37464, 2396],
     [31030, 1993], [24843, 1627], [36172, 2375], [39552, 2560],
     [72545, 4597], [75352, 4871], [18031, 1119], [36961, 2503],
     [43621, 2992], [15694, 1042], [36231, 2487], [29945, 2014],
```

```
[40588, 2805], [75255, 5062], [37709, 2643], [30899, 2126],
[25486, 1784], [37497, 2641], [40398, 2766], [74105, 5047],
[76725, 5312], [18317, 1215]
])
```

We could also read the datasets from files, using the svmlight format for instance, where each observation is described by one line with the structure:

```
<class-label> <feature-idx>:<value> <feature-idx>:<value> ...
```

For the data set in Table 5.8, the corresponding file consists of the following lines:

```
0 1:35680 2:2217
0 1:42514 2:2761
0 1:15162 2:990
...
1 1:36961 2:2503
1 1:43621 2:2992
1 1:15694 2:1042
...
```

that we load with this piece of code:

```
from sklearn.datasets import load_svmlight_file

X, y = load_svmlight_file(file)
```

We then select and fit a model, here logistic regression with default parameters, with the lines:

```
classifier = linear_model.LogisticRegression()
model = classifier.fit(X, y)
```

Once the model is trained, we can apply it to new observations. The next instruction just reapplies it to the training set:

```
y_predicted = classifier.predict(X)
```

which predicts exactly the classes we had in this set:

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

5.12.2 Evaluating a Model

In our previous experiment, we trained and tested a model on the same dataset. This is not a good practice: A model that would just memorize the data would also reach a perfect accuracy. As we want to be able to generalize beyond the training set to new data, the standard evaluation procedure is to estimate the performance on a distinct and unseen test set. When we have only one set, as in Table 5.8, we can divide it in two subsets: The training set and the test set (or holdout data). The split can be 90% of the data for the training set and 10% for the test set or 80–20.

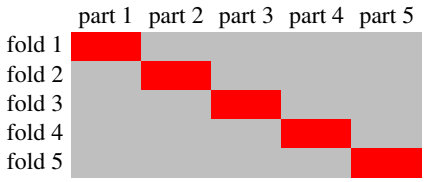

Fig. 5.17. Five-fold cross validation with five different partitions. In each fold, the test set is in red and the training set in gray

In the case of a small dataset like ours, a specific test set may lead to optimize the classifier just for it and hence create an overfit. Cross validation, or *N*-fold cross validation, is a technique to mitigate such an overfit. Instead of using one single test set, cross validation uses multiple splits called the folds. In a five-fold cross validation, the evaluation is carried out on five different test sets randomly sampled from the data set. In each fold, the rest of the data set serves as training set. Figure 5.17 shows a 5-fold cross validation process, where we partitioned the dataset into five equal sized subsets. In each fold, one of the subsets is the test set (red) and the rest, the training set (gray). The evaluation is then repeated five times with different test sets and the final result is the mean of the results of the five different folds.

The number of folds depends on the size of the dataset and the computing resources at hand: 5 and 10 being frequent values. At the extreme, a leave-one-out cross-validation has as many folds as there are observations. At each fold, the training set consists of all the observations except one, which is used as test set.

scikit-learn has built-in cross validation functions. The code below shows an example of it with a 5-fold cross validation and a score corresponding to the accuracy:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(classifier, X, y, cv=5,
                         scoring='accuracy')
print('Score', scores.mean())
```

5.12.3 Nominal Data

The scikit-learn's classifiers use numerical algorithms that we cannot apply to nominal data such as those in Table 5.7, where **X** and **y** correspond to:

$$\mathbf{X} = \begin{bmatrix} \text{Sunny} & \text{Hot} & \text{High} & \text{False} \\ \text{Sunny} & \text{Hot} & \text{High} & \text{True} \\ \text{Overcast} & \text{Hot} & \text{High} & \text{False} \\ \text{Rain} & \text{Mild} & \text{High} & \text{False} \\ \text{Rain} & \text{Cool} & \text{Normal} & \text{False} \\ \text{Rain} & \text{Cool} & \text{Normal} & \text{True} \\ \text{Overcast} & \text{Cool} & \text{Normal} & \text{True} \\ \text{Sunny} & \text{Mild} & \text{High} & \text{False} \\ \text{Sunny} & \text{Cool} & \text{Normal} & \text{False} \\ \text{Rain} & \text{Mild} & \text{Normal} & \text{False} \\ \text{Sunny} & \text{Mild} & \text{Normal} & \text{True} \\ \text{Overcast} & \text{Mild} & \text{High} & \text{True} \\ \text{Overcast} & \text{Hot} & \text{Normal} & \text{False} \\ \text{Rain} & \text{Mild} & \text{High} & \text{True} \end{bmatrix} ; \mathbf{y} = \begin{bmatrix} \text{N} \\ \text{N} \\ \text{P} \\ \text{P} \\ \text{P} \\ \text{N} \\ \text{P} \\ \text{N} \\ \text{P} \\ \text{P} \\ \text{P} \\ \text{P} \\ \text{P} \\ \text{N} \end{bmatrix}$$

We need first to convert this dataset into binary vectors as we saw in Sect. 5.11. This can be done with the `DictVectorizer` class that transforms lists of dictionaries representing the observations into vectors.

Let us first read the dataset assuming the file consists of values separated by commas (CSV). We can use the csv module library and `DictReader()` that creates a list of dictionaries from the dataset. The column names are given in the `fieldnames` parameter:

```
import csv

column_names = ['outlook', 'temperature', 'humidity',
                'windy', 'play']
dataset = list(csv.DictReader(open('weather-nominal.csv'),
                              fieldnames=column_names))
```

We create the `X_dict` and `y_symbols` tables from `dataset`, where we use a deep copy to preserve the dataset:

```
import copy

def extract_feats(dataset, class_name):
    X_dict = copy.deepcopy(dataset)
    y_symbols = [obs.pop(class_name, None) for obs in X_dict]
    return X_dict, y_symbols

X_dict, y = extract_feats(dataset, 'play')
```

Now we have all our data in two separate tables. We need to convert `X_dict` into a numeric matrix and we use `DictVectorizer` to carry this out:

```
from sklearn.feature_extraction import DictVectorizer
```

```
vec = DictVectorizer(sparse=False) # Should be true
X = vec.fit_transform(X_dict)
```

that returns:

```
array([[ 1.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  1.,  0.],
       [ 1.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  1.,  0.],
       [ 1.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  1.,  0.],
       [ 0.,  1.,  0.,  1.,  0.,  1.,  0.,  0.,  1.,  0.],
       [ 0.,  1.,  0.,  1.,  0.,  1.,  0.,  0.,  0.,  1.],
       [ 0.,  1.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  1.],
       [ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  1.,  1.,  0.],
       [ 0.,  1.,  0.,  0.,  1.,  1.,  0.,  0.,  1.,  0.],
       [ 0.,  1.,  0.,  1.,  0.,  0.,  0.,  1.,  1.,  0.],
       [ 0.,  1.,  0.,  0.,  1.,  0.,  0.,  1.,  0.,  1.],
       [ 1.,  0.,  1.,  0.,  0.,  0.,  0.,  1.,  0.,  1.],
       [ 0.,  1.,  1.,  0.,  0.,  0.,  1.,  0.,  1.,  0.],
       [ 1.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  1.]])
```

something very similar to Table 5.10. We set the `sparse` parameter to `False` to be be able to visualize the matrix. In most cases, it should be set to `True` to save space.

We can then use any scikit-learn classifier to train a model and predict classes.

5.13 Neural Networks

Neural networks form a last family of numerical classifiers that, compared to the other techniques we have seen, have a more flexible and extendible architecture. Typically, neural networks are composed of layers, where each layer contains a set of nodes, the neurons. The input layer corresponds to the input features, where each feature is represented by a node, and the output layer produces the classification result.

Beyond this simple outline, there are scores of ways to build and configure a neural net in practice. In this chapter, we will focus on the simplest architecture, **feed-forward**, that consists of a sequence of layers. In a given layer, each node receives information from the all the nodes of the preceding layer, processes this information, and passes it through to all the nodes of the next layer; see Fig. 5.18.

Over the years, neural networks have become one the most efficient machine-learning devices. In this section, we will describe how we can reformulate the perceptron and logistic regression as neural networks, and then see how we can extend the networks with multiple layers. In the next chapters, we will introduce other types of neural networks.

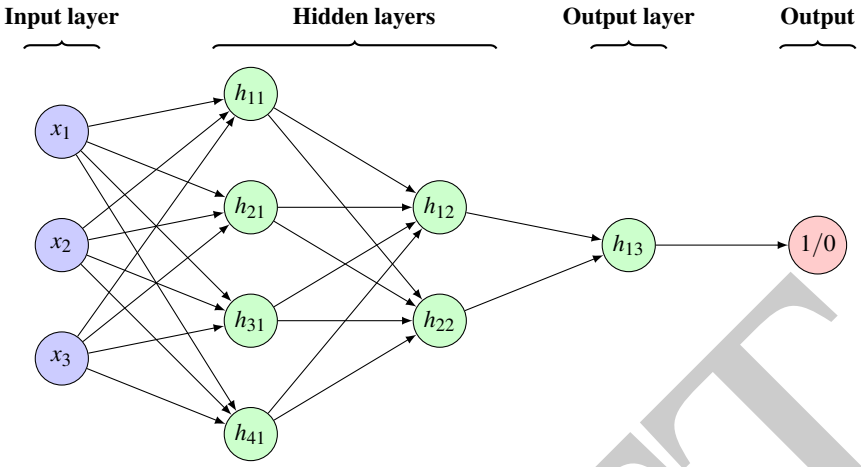

Fig. 5.18. Overview of a neural network

5.13.1 Architecture

The structure of neural networks was initially inspired by that of the brain and its components, the nerve cells. Literature in the field often uses a biological vocabulary to describe the network components: The connections between neurons are then called synapses and the information flowing between two nodes, a scalar number, is multiplied by a weight called the synaptic weight. At a given layer, the computation carried out in a neuron has two main steps:

1. Combine information coming from the neurons from the preceding layer; this is simply done through the dot product of the synaptic weights. In Fig 5.18, let us call $\mathbf{x}$, the input vector and $\mathbf{w}_{11}$, the weights from the incoming synapses at node h_{11}. The linear combination is: $\mathbf{w}_{11} \cdot \mathbf{x}$.
   To represent the weights of all the incoming connections of the first hidden layer, we use a matrix, $\mathbf{W}_{h1}$, where we store the weights as rows. There are as many rows as there are hidden nodes in the layer. We compute the input values of all the nodes by multiplying the matrix by $\mathbf{x}$:

   $$\mathbf{W}_{h1} \cdot \mathbf{x},$$

   In the case of Fig. 5.18, we stack the $\mathbf{w}_{11}, \mathbf{w}_{21}, \mathbf{w}_{31}$, and $\mathbf{w}_{41}$ vectors as rows of the matrix. In most implementations, the dot product includes an intercept or bias:

   $$\mathbf{W}_{h1} \cdot \mathbf{x} + \mathbf{b},$$

   which is not represented in Fig 5.18;
2. Pass the resulting numbers, a vector, to an **activation function**. This produces the output of the neurons. For the first layer, we have:

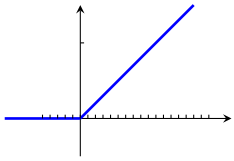

Fig. 5.19. The reLU function

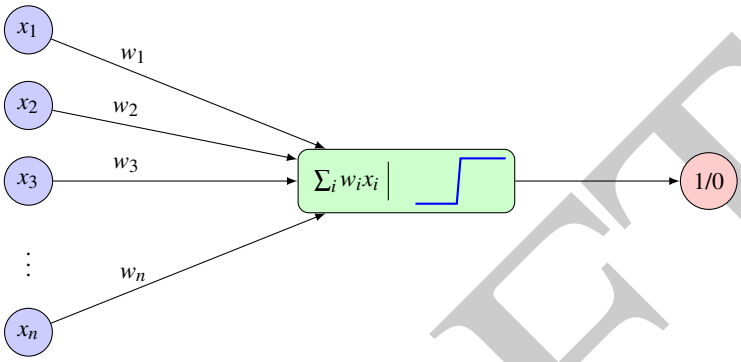

Fig. 5.20. Neural network representation of the perceptron

$$\text{activation}(\mathbf{W}_{h1} \cdot \mathbf{x} + \mathbf{b}).$$

The four most common functions used as activation are: The Heaviside, logistic, hyperbolic tangent functions, that we have already seen, and the rectified linear unit (reLU) function, Fig. 5.19, where:

$$reLU(x) = \max(0, x).$$

5.13.2 The Perceptron and Logistic Regression

The perceptron from Sect. 5.8 is the most simple form of neural networks. It has a single layer, where the activation is the Heaviside function. Figure 5.20 shows a graphical representation of it.

We can also reformulate logistic regression from Sect. 5.10 as a neural network. It also has a single layer, where the activation function is the logistic function; see Figure 5.21.

5.13.3 Hidden Layers

From the examples above, we can build more complex networks by just adding layers between the input features and the layer connected to the output. A frequent design

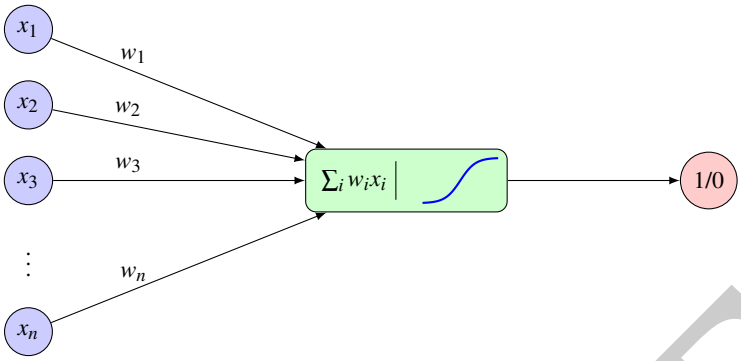

Fig. 5.21. Neural network representation of logistic regression

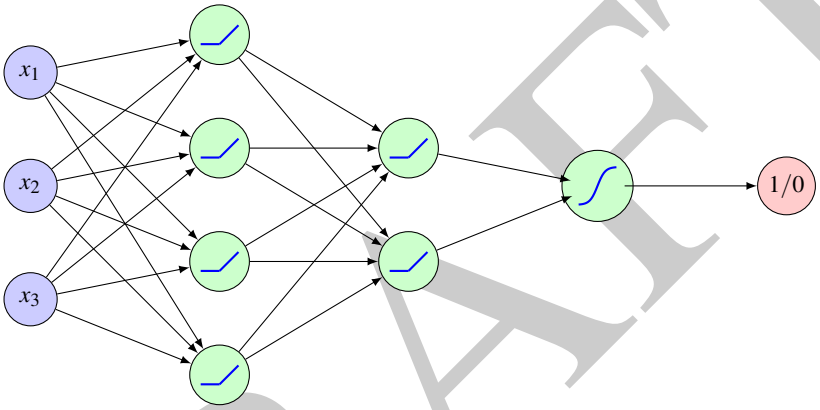

Fig. 5.22. Network with hidden layers and two activation functions: reLU and logistic regression

is to use the reLU function in the hidden layers, except the last one, where we use the logistic function; see Fig. 5.22.

Training a neural network model is then identical to what we have seen for the perceptron or logistic regression. Given a topology and activation functions, we apply a training algorithm to a dataset that finds all the synaptic weights, i.e. the weight vectors, of the network. To carry this out, the training algorithms use a generalized form of gradient descent called **backpropagation**, where the errors in the last layer are back propagated to adjust the weights iteratively.

5.14 Programming Neural Networks

We will now use the concepts we have learned in a program that we will apply to the *Salammbô* dataset. For our implementation, we will use Keras (Chollet, 2018) and

Tensorflow (Abadi et al., 2016), two comprehensive libraries. Tensorflow contains most neural network algorithms we need and Keras is a high-level programming interface on top of Tensorflow.

5.14.1 Building the Network

Building a feed-forward network is easy with Keras. We just need to describe its structure in terms of layers. Feed-forward networks correspond to the `Sequential` class and fully interconnected nodes to `Dense` layers. To implement logistic regression, we then use a `Sequential` model and one `Dense()` layer. In the `Dense()` object, we specify the output dimension as first parameter, 1, for English (0) or French (1), then, using named arguments, the input dimension, `input_dim = 2`, for the total number of characters and of *A*s, and finally the activation function, a sigmoid, `activation='sigmoid'`. To have a reproducible model, we also set a random seed. Creating our network is as simple as that:

```
from keras import models
from keras import layers

np.random.seed(0)
model = models.Sequential()
model.add(layers.Dense(1, input_dim=2, activation='sigmoid'))
```

5.14.2 Data Representation and Preprocessing

We store the *Salammbô* dataset, **X** and **y**, in numpy arrays using the same structure as in Sect. 5.12.1. The algorithms in Tensorflow are very sensitive to differences in numeric ranges between the features. This means that prior to fitting the model, we need to standardize the counts for each character by subtracting the mean from the counts and dividing them by the standard deviation:

$$x_{i,j_{std}} = \frac{x_{i,j} - \bar{x}_{.,j}}{\sigma_{x_{.,j}}}.$$

For letter A, the second column in the **X** matrix, we compute the mean and standard deviation of the counts with these two statements:

```
mean = np.mean(X[:,1])
std = np.std(X[:,1])
```

where the results are 2716.5 and 1236.21. Applying a standardization replaces the value of $x_{15,1}$, 2503, with -0.1727.

The chapters in *Salammbô* have different sizes: The largest chapter is five times the length of the shorter. To mitigate the count differences, we can also apply a normalization before the standardization and divide the chapter vectors by their norm. This will constrain the chapters to have a unit norm:

$$x_{i,j_{norm}} = \frac{x_{i,j}}{\sqrt{\sum_{k=0}^{n-1} x_{i,k}^2}}.$$

Instead of computing the mean and standard deviation ourself, we rely on scikit-learn and its built-in classes, `Normalizer` and `StandardScaler`, to normalize and standardize an array. The code is straightforward with the `fit()` and `transform()` methods:

```
from sklearn.preprocessing import StandardScaler, Normalizer

X_norm = Normalizer().fit_transform(X)
X_scaled = StandardScaler().fit_transform(X_norm)
```

where `fit_transform()` is the sequence of `fit()` and `transform()`.

5.14.3 Training the Model

Before we can train the model, we need to specify the loss function (the quantity to minimize) and the algorithm to compute the weights. We do this with the `compile()` method and two arguments: `loss` and `optimizer`, where we use, respectively, `binary_crossentropy` for logistic regression and `rmsprop`, a variant of the plain gradient descent. We also tell the model to report the accuracy.

```
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Once the model is compiled, we train it with the `fit()` method, where we pass the scaled data set, the number of epochs, and the batch size. If we want a purely stochastic descent, we set `batch_size` to 1. In most cases, the batch size should have higher values.

```
model.fit(X_scaled, y, epochs=10, batch_size=1)
```

Once trained, we can apply the model with the `predict()` method, here to the training set again. We can also evaluate it using `evaluate` and report the cross entropy and the accuracy:

```
y_predicted = model.predict(X_scaled)
# evaluate the model
scores = model.evaluate(X_scaled, y)
```

In this small example, where we train and apply a model on the same dataset, we reach an accuracy of 100%.

5.14.4 Adding Hidden Layers

So far, we used a single layer. If we want to add hidden layers, we just use the `add()` method, for instance to insert a layer of 10 nodes. As mentioned earlier, intermediate layers should use the `relu` activation function. We do not need to specify the input size; Keras automatically infers it:

```
model.add(layers.Dense(10, input_dim=2, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Training and running this new network will also result in a 100% accuracy.

This last example shows how to build a multilayer feedforward network. However, this architecture is not realistic here given the size of the dataset: The high number of nodes relative to the size certainly creates an overfit.

5.15 Further Reading

Information theory is covered by many books, many of them requiring a good mathematical background. The text by Manning and Schütze (1999, Chap. 2) provides a short and readable introduction oriented toward natural language processing.

Machine-learning techniques are now ubiquitous in all the fields of natural language processing. ID3 outputs classifiers in the form of decision trees that are easy to understand. It is a simple and robust algorithm. Logistic regression, perceptrons, support vector machines, and neural networks are other popular classifiers. Which one to choose has no easy answer as they may have different performances on different data sets. When starting an analysis, my preferences are leaning toward ID3 and logistic repression to have a baseline, although this does not exclude the others. Supervised machine-learning is a large and evolving domain. In this chapter, we set aside many details and techniques. Hastie et al. (2009), Saporta (2011), Murphy (2012), James et al. (2013), and Goodfellow et al. (2016) are mathematical references on classification and statistical learning in general that can complement this chapter. Grus (2015) is a pedagogical description on how to implement machine-learning algorithms in Python and Chollet (2018), a very clear programming introduction to deep learning.

We used regression to introduce linear classification techniques. This line-fitting process has a somehow enigmatic name. It is due to Galton (1886) who modeled the transmission of stature from parents to children. He gathered a data set of the heights of children and parents and observed that taller-than-average parents tended to have children shorter than they, and that shorter parents tended to have taller children than they. Galton called this a *regression towards mediocrity*.

Character statistics, as in Sect 5.4, form the basis of language detection. However, the detection algorithms usually consider the counts of all the characters of a language instead of one single character, A. The algorithms also often extend to sequences of two and three characters. An example of modern language detector is given by the Compact Language Detector v3 (CLD3) from Google,

`https://github.com/google/cld3`, that uses a simple feed-forward neural network with one hidden layer.

A number of machine-learning toolkits are available from the Internet. Scikit-learn (Pedregosa et al., 2011) is a comprehensive collection of machine-learning and data analysis algorithms with an API in Python (`http://scikit-learn.org/`). Keras (Chollet, 2018) is a neural network library also in Python. We will use these two libraries in this book. R is another set of statistical and machine-learning functions with a script language (`http://www.r-project.org/`). Weka (Witten and Frank, 2005; Hall et al., 2009) is collection of machine-learning algorithms written in Java (`http://www.cs.waikato.ac.nz/ml/weka/`). Deeplearning4j (Patterson and Gibson, 2017) is a library for neural networks, also in Java.

Exercises

5.1. Implement the ID3 algorithm in Python, or another language. Test it on the data set in Table 5.7.

5.2. Implement linear regression using the gradient descent. Test it on the data set in Table 5.8 with, respectively, English and French.

5.3. Implement the perceptron algorithm. Test it on the data set in Table 5.8.

5.4. Implement logistic regression. Test it on the data set in Table 5.8.

5.5. In this exercise, you will improve the language detector of Sects. 5.12 and 5.14:

1. Collect a corpus of texts in two languages, for instance French and English. You should have at least one-million characters per language;
2. Divide your dataset in a training set and a test set, respectively 80% and 20% of the data;
3. For each text in your corpus, compute the statistics of all the letters and present them as in Table 5.8 for the training and test sets;
4. Train and evaluate a language detector based on logistic regression and scikit-learn;
5. Train and evaluate a language detector based on a neural network with one hidden-layer. You will use Keras this time.

6

Counting and Indexing Words

6.1 Text Segmentation

Most language processing techniques, such as language modeling and morphological and syntactic parsing, consider words and sentences. When the input data is a stream of characters, we must first segment it, i.e., identify the words and sentences in it, before we can apply any further operation to the text. We call this step **text segmentation** or **tokenization**. A tokenizer can also remove formatting instructions, such as XML tags, if any.

Originally, early European scripts had no symbols to mark segment boundaries inside a text. Ancient Greeks and Romans wrote their inscriptions as continuous strings of characters flowing from left to right and right to left without punctuation or spaces. The *lapis niger*, one of the oldest remains of the Latin language, is an example of this writing style, also called **boustrophedon** (Fig. 6.1).

As the absence of segmentation marks made texts difficult to read, especially when engraved on a stone, Romans inserted dots to delimit the words and thus improve their legibility. This process created the graphic word as we know it: a sequence of letters between two specific signs. Later white spaces replaced the dots as word boundaries and Middle Ages scholars introduced a set of punctuation signs: commas, full stops, question and exclamation marks, colons, and semicolons, to delimit phrases and sentences.

6.1.1 What Is a Word?

The definition of what a word is, although apparently obvious, is in fact surprisingly difficult. A naïve description could start from its historical origin: a sequence of alphabetic characters delimited by two white spaces. This is an approximation. In addition to white spaces, words can end with commas, question marks, periods, etc. Words can also include dashes and apostrophes that, depending on the context, have a different meaning.

Word boundaries vary according to the language and orthographic conventions. Compare these different spellings: *news stand*, *news-stand*, and *newsstand*. Although

Fig. 6.1. Latin inscriptions on the *lapis niger*. *Corpus inscriptionum latinarum*, CIL I, 1. Picture from Wikipedia

the latter one is considered more correct, the two other forms are also frequent. Compare also the convention in German to bind together adjacent nouns as in *Gesundheitsreform*, as opposed to English that would more often separate them, as in *health reform*. Compare finally the ambiguity of punctuation marks, as in the French word *aujourd'hui*, 'today', which forms a single word, and *l'article*, 'the article', where the sequence of an article and a noun must be separated before any further processing.

In corpus processing, text elements are generally called **tokens**. Tokens include words and also punctuation, numbers, abbreviations, or any other similar type of string. Tokens may mix characters and symbols as:

- Numbers: 9,812.345 (English and French from the 18th–19th century), 9 812,345 (current French and German) 9.812,345 (French from the 19th–early 20th century);
- Dates: 01/02/2003 (French and British English), 02/01/2003 (US English), 2003/02/01 (Swedish);
- Abbreviations and acronyms: km/h, m.p.h., S.N.C.F.;
- Nomenclatures: A1-B45, /home/pierre/book.tex;
- Destinations: Paris–New York, Las Palmas–Stockholm, Rio de Janeiro–Frankfurt am Main;
- Telephone numbers: (0046) 46 222 96 40;
- Tables;
- Formulas: $E = mc^2$.

As for the words, the definition of what is a sentence is also tricky. A naïve definition would be a sequence of words ended by a period. Unfortunately, periods are also ambiguous. They occur in numbers and terminate abbreviations, as in *etc.* or *Mr.*, which makes sentence isolation equally complex. In the next sections, we examine techniques to break a text into words and sentences, and to count the words.

6.1.2 Breaking a Text into Words and Sentences

Tokenization breaks a character stream, that is, a text file or a keyboard input, into tokens – separated words – and sentences. In Python, it results in a list of strings. For this paragraph, such a list looks like:

```
[['Tokenization', 'breaks', 'a', 'character', 'stream', ',',
'that', 'is', ',', 'a', 'text', 'file', 'or', 'a', 'keyboard',
'input', ',', 'into', 'tokens', '-', 'separated', 'words', '-',
'and', 'sentences', '.'], ['In', 'Python', ',', 'it', 'results',
'in', 'a', 'list', 'of', 'strings', '.'], ['For', 'this',
'paragraph',  ',', 'such', 'a', 'list', 'looks', 'like', ':']]
```

An basic format to output or store tokenized texts is to print one word per line and have a blank line to separate sentences as in:

```
In
Python
,
it
results
in
a
list
of
atoms
.

For
this
paragraph
,
such
a
list
looks
like
:
```

6.2 Tokenizing Words

We now introduce tokenization techniques using two complementary approaches. The first one considers the unit boundaries, and the second one their content. We will then merge them into a more elaborate program in Python.

6.2.1 Using White Spaces

Tokenizing texts using white spaces as word delimiters is the most elementary technique. It is straightforward in Python, as shown in the program below: we just replace sequences of white spaces in the text with a new line, and we consider what is between two white spaces to be a word. In the program, we use the `\s` character class to represent the white space:

```
import re

one_token_per_line = re.sub('\s+', '\n', text)
```

However, this does not work perfectly as with the first lines of the *Odyssey*:

> Tell me, O muse, of that ingenious hero who travelled far and wide after he had sacked the famous town of Troy.

Let us assign this sentence to the `text` variable:

```
text = """Tell me, O muse, of that ingenious hero who
travelled far and wide after he had sacked the famous
town of Troy."""
```

and run the program. We obtain:

```
Tell
me,
O
muse,
of
that
ingenious
hero
...
```

where the commas are not segmented from the words.

6.2.2 Using White Spaces and Punctuation

The previous program failed to tokenize the punctuation. To improve it, we need to process separately the punctuation and symbols. We identify these characters with regular expressions and we insert white spaces around them to separate them from the words. The punctuation and symbols we want to tokenize include:

1. The dot: `.`
2. Other boundary signs: `,;:?!#%&-/\`
3. Brackets: `"()[]{}`,
4. Quotes: `‘‘ ‘ ’’ ’`, and
5. Symbols: `$<>`.

Table 6.1. Apostrophe tokenization in English

Contracted form	Example	Tokenization	Expanded form
'm	*I'm*	`I 'm`	*I am*
'd	*we'd*	`we 'd`	*we had* or *we would*
'll	*we'll*	`we 'll`	*we will*
're	*you're*	`you 're`	*you are*
've	*I've*	`I 've`	*I have*
n't	*can't*	`can n't`	*cannot*
's	*she's*	`she 's`	*she has* or *she is*
's	*Pierre's book*	`Pierre 's book`	Possessive marking

We could write a list of all these characters. It is easier and more compact to use the Unicode punctuation and symbol classes instead: `\p{P}` and `\p{S}`, respectively; see Table 4.6 for a complete list. To use the Unicode classes, we need to import the `regex` module.

We then tokenize the text according to white spaces as in the previous section:

```
import regex as re

spaced_tokens = re.sub('([\p{S}\p{P}])', r' \1 ', text)
one_token_per_line = re.sub('\s+', '\n', spaced_tokens)
```

Running this code on our small text results in:

```
Tell
me
,
O
muse
,
of
that
ingenious
hero
who
travelled
far
...
```

This second program produces a better result than our first one, although not perfect. Decimal numbers, for example, would not be properly processed. The program would match the point of decimal numbers such as 3.14 and insert new lines between 3 and 14. The apostrophe inside words is another ambiguous sign. The tokenization of auxiliary and negation contractions in English is unpredictable without a morphological analysis. It requires a dictionary with all the forms (Table 6.1).

In French, apostrophes corresponding to the elided *e* have a regular behavior as in

Si j'aime et d'aventure → `si j' aime et d' aventure`

but there are words like *aujourd'hui*, 'today', that correspond to a single entity and are not tokenized.

6.2.3 Returning a List of Tokens

Our tokenizers, so far, returned a string, where we inserted new lines at the end of each token to print one token per line. If we want to have the tokens in the form of a list instead, we can use `split()` that breaks a string into pieces. The split points are defined by a delimiter and the resulting tokens, the substrings between the delimiters, are assigned sequentially to a list.

In Python, `split()` has two variants:

- `str.split(separator)`, where the separator is a constant string. If there is no argument, the default separator is a sequence of white spaces, including tabulations and new lines;
- `re.split(pattern, string)`, where the separator is a regular expression, `pattern`, that breaks up the `string` variable as many times as `pattern` matches in `string`.

We can apply either method to our tokenized string `one_token_per_line`:

```
tokens = one_token_per_line.split()
```

or

```
tokens = re.split('\s+', one_token_per_line)
```

Both produce a list of tokens:

```
['Tell', 'me', ',', 'O', 'muse', ',', 'of', 'that',
'ingenious', 'hero', 'who', 'travelled', 'far', ...]
```

`re.split(pattern, string)` creates empty strings when the separator is at the beginning or the end of `string`. We remove them with the `remove('')` list method.

6.2.4 Defining Contents

Alternatively, we can explicitly define the content of words. If we just want to extract a sequence of words, where a word is defined as a sequence of contiguous letters, we can use the `re.findall()` function and the `\p{L}+` pattern. `re.findall()` returns a list of words:

```
import regex as re

re.findall('\p{L}+', text)
```

resulting in:

```
['Tell', 'me', 'O', 'muse', 'of', 'that', 'ingenious',
'hero', ...]
```

We can see here that the Unicode classes are considerably more concise than the exhaustive enumeration of their components. Should we want to list explicitly the letters used in English, French, German, and Swedish, we would need this class, probably forgetting some characters:

```
[a-zàâäæçéèêëîïôöœùûüÿßA-ZÀÂÄÆÇÉÈÊËÎÏÔÖŒÙÛÜŸ]+
```

If we want to extend the tokens to punctuation and symbols, we need to isolate them on a single line as in Sect. 6.2.2. We then return a list of tokens with `findall()`. We ignore all the other characters. The short program:

```
spaced_tokens = re.sub('([\p{S}\p{P}])', r' \1 ', text)
re.findall('[\p{S}\p{P}\p{L}]+', spaced_tokens)
```

yields:

```
['Tell', 'me', ',', 'O', 'muse', ',', 'of', 'that',
'ingenious', 'hero', 'who', 'travelled', 'far', ...]
```

6.2.5 Tokenizing Using Classifiers

So far, we have carried out tokenization using rules that we have explicitly defined and implemented using regular expressions or Python. A second option is to use classifiers such as logistic regression (Sect. 5.10) and to train a tokenizer from a corpus. Given an input queue of characters, we then formulate tokenization as a binary classification: is the current character the end of a token or not? If the classifier predicts a token end, we insert a new line.

Before we can train our classifier, we need a corpus and an annotation to mark the token boundaries. Let us use the OpenNLP format as an example. The Apache OpenNLP library is an open-source toolkit for natural language processing. It features a classifier-based tokenizer and has defined an annotation for it (Apache OpenNLP Development Community, 2012). A training corpus consists of a list of sentences with one sentence per line, where the white spaces are unambiguous token boundaries. The other token boundaries are marked with the `<SPLIT>` tag, as in these two sentences:

```
Pierre Vinken<SPLIT>, 61 years old<SPLIT>, will join the
    board as a nonexecutive director Nov. 29<SPLIT>.
Mr. Vinken is chairman of Elsevier N.V.<SPLIT>, the Dutch
    publishing group<SPLIT>.
```

Note that in the example above, the sentence lengths are too long to fit the size of the book and we inserted two additional breaks and leading spaces to denote a continuing sentence. In the corpus file, every new line corresponds to a new sentence.

Context	Current char.	Previous pair	Next char.	Next pair	Class	Action
*Vinke**n**,*	*n*	*en*	,	,␣	Token end	New line
*ol**d**,*	*d*	*ld*	,	,␣	Token end	New line
*No**v**.*	*v*	*ov*	.	.␣	Inside token	Nothing

Table 6.2. The features extracted from the second *n* in *Pierre Vinken*, the *d* in *old*, and the dot in *Nov.*. The two classes to learn are **inside token** and **token end**

Once we have an annotation, we need to define the features we will use for the classifier. We already used features in Sect. 5.4 in the form of letter frequencies to classify the language of a text. For the tokenization, we will follow Reynar (1998, pp. 69–70), who describes a simple feature set consisting of four features:

- The current character,
- The pair formed of the previous and current characters,
- The next character,
- The pair formed of the two next characters.

As examples, Table 6.2 shows the features extracted from three characters in the sentences above: the second *n* in *Pierre Vinken*, the *d* in *old*, and the dot in *Nov.* From these features, the classifier will a create model and discriminate between the two classes: **inside token** and **token end**.

Before we can learn the classifiers, we need a corpus annotated with the `<SPLIT>` tags. We can create one by tokenizing a large text manually – a tedious task – or by reconstructing a nontokenized text from an already tokenized text. See, for example, the Penn Treebank (Marcus et al., 1993) for English.

We extract a training data set from the corpus by reading all the characters and extracting for each character their four features and their class. We then train the classifier, for instance, using logistic regression, to create a model. Finally, given a nontokenized text, we apply the classifier and the model to each character of the text to decide if it is inside a token or if it is a token end.

6.3 Sentence Segmentation

6.3.1 The Ambiguity of the Period Sign

Sentences usually end with a period, and we will use this sign to recognize boundaries. However, this is an ambiguous symbol that can also be a decimal point or appear in abbreviations or ellipses. To disambiguate it, we introduce now two main lines of techniques identical to those we used for tokenization: rules and classifiers.

Although in this chapter, we describe sentence segmentation after tokenization, most practical systems use them in a sequence, where sentence segmentation is the first step followed by tokenization.

Table 6.3. Recognizing numbers. After Grefenstette and Tapanainen (1994)

Fractions, dates	`[0-9]+(\/[0-9]+)+`
Percent	`([+\-])?[0-9]+(\.)?[0-9]*%`
Decimal numbers	`([0-9]+,?)+(\.[0-9]+|[0-9]+)*`

6.3.2 Rules to Disambiguate the Period Sign

We will consider that a period sign either corresponds to a sentence end, a decimal point, or a dot in an abbreviation. Most of the time, we can recognize these three cases by examining a limited number of characters to the right and to the left of the sign. The objective of disambiguation rules is then to describe for each case what can be the left and right context of a period.

The disambiguation is easier to implement as a two-pass search: the first pass recognizes decimal numbers or abbreviations and annotates them with a special marking. The second one runs the detector on the resulting text. In this second pass, we also include the question and exclamation marks as sentence boundary markers.

We can generalize this strategy to improve the sentence segmentation with specific rules recognizing dates, percentages, or nomenclatures that can be run as different processing stages. However, there will remain cases where the program fails, notably with abbreviations.

6.3.3 Using Regular Expressions

Starting from the most simple rule to identity sentence boundaries, a period corresponds to a full stop, Grefenstette and Tapanainen (1994) experimented on a set of increasingly complex regular expressions to carry out segmentation. They evaluated them on the Brown corpus (Francis and Kucera, 1982).

About 7% of the sentences in the Brown corpus contain at least one period, which is not a full stop. Using their first rule, Grefenstette and Tapanainen could correctly recognize 93.20% of the sentences. As a second step, they designed the set of regular expressions in Table 6.3 to recognize numbers and remove decimal points from the list of full stops. They raised to 93.78% the number of correctly segmented sentences.

Regular expressions in Table 6.3 are designed for English text. French and German decimal numbers would have a different form as they use a comma as decimal point and a period or a space as a thousand separator:

```
([0-9]+(.| )?)*[0-9](,[0-9]+)
```

Finally, Grefenstette and Tapanainen added regular expressions to recognize abbreviations. They used three types of patterns:

- A single capital followed by a period as *A.*, *B.*, *C.*
- A sequence of letters and periods as in *U.S.*, *i.e.*, *m.p.h.*,
- A capital letter followed by a sequence of consonants as in *Mr.*, *St.*, *Ms.*

Table 6.4. Regular expressions to recognize abbreviations and performance breakdown. The *Correct* column indicates the number of correctly recognized instances, *Errors* indicates the number of errors introduced by the regular expression, and *Full stop* indicates abbreviations ending a sentence where the period is a full stop at the same time. After Grefenstette and Tapanainen (1994)

Regex	Correct	Errors	Full stop
`[A-Za-z]\.`	1,327	52	14
`[A-Za-z]\.([A-Za-z0-9]\.)+`	570	0	66
`[A-Z][bcdfghj-np-tvxz]+\.`	1,938	44	26
Totals	3,835	96	106

Table 6.4 shows the corresponding regular expressions as well as the number of abbreviations they recognize and the errors they introduce. Using them together with the regular expressions to recognize decimal numbers, Grefenstette and Tapanainen could increase the correct segmentation rate to 97.66%.

6.3.4 Improving the Tokenizer Using Lexicons

Grefenstette and Tapanainen (1994) further improved their tokenizer by automatically building an abbreviation lexicon from their corpus. To identify potential abbreviations, they used the following idea: a word ending with a period that is followed by either a comma, a semicolon, a question mark, or a lowercase letter is a likely abbreviation. Grefenstette and Tapanainen (1994) applied this idea to their corpus; however, as they gathered many words that were not abbreviations, they removed all the strings in the list that appeared without a trailing period somewhere else in the corpus. They then reached 98.35%.

Finally, using a lexicon of words and common abbreviations, *Mr.*, *Sen.*, *Rep.*, *Oct.*, *Fig.*, *pp.*, etc., they could recognize 99.07% of the sentences. Mikheev (2002) describes another efficient method that learns tokenization rules from the set of ambiguous tokens distributed in a document. While most published experiments have been conducted on English, Kiss and Strunk (2006) present a multilingual statistical method that can be trained on unannotated corpora.

6.3.5 Sentence Detection Using Classifiers

As for tokenization, we can use classifiers, such as decision trees or logistic regression, to segment sentences. The idea is simple: given a period in a text (or a question or an exclamation mark), classify it as the end of a sentence or not. The implementation is identical to that in Sect. 6.2.5 and we can use the same corpus: we just ignore the `<SPLIT>` tags.

Practically, we need to collect a data set and define the features to associate to the periods. Reynar and Ratnaparkhi (1997) proposed a method that we describe here. As corpus, they used the Penn Treebank (Marcus et al., 1993), from which they

Context	Prefix	Suffix	Previous word	Next word	Prefix abbrev.	Class
Nov.	*Nov*	nil	*director*	*29.*	Yes	Inside sentence
29.	*29*	nil	*Nov.*	*Mr.*	No	End of sentence

Table 6.5. The features extracted from *Nov.* and *29.* in the example sentences in Sect. 6.2.5. The two classes to learn are **inside sentence** and **end of sentence**

extracted all the strings separated by white spaces and containing a period. They used a compact set of eight features:

1. The characters in the string to the left of the period (the prefix);
2. The character to the right of the period (the suffix);
3. The word to the left of the string;
4. The word to the right of the string;
5. Whether the prefix (resp. suffix) is on a list of abbreviations;
6. Whether the word to the left (resp. to the right) is on a list of abbreviations.

Table 6.5 shows the features for the periods in *Nov.* and *29.* in the example sentences in Sect. 6.2.5. The first four features are straightforward to extract. We need a list of abbreviations for the rest. We can build this list automatically using the method described in Sect. 6.3.4.

Reynar and Ratnaparkhi (1997) used logistic regression to train their classification models and discriminate between the two classes: **inside sentence** and **end of sentence**.

6.4 Word Counting

6.4.1 Some Definitions

The first step of lexical statistics consists in extracting the list of **word types** or **types**, i.e., the distinct words, from a corpus, along with their frequencies. Within the context of lexical statistics, word types are opposed to word tokens, the sequence of running words of the corpus. The excerpt from George Orwell's *Nineteen Eighty-Four*:

> War is peace
> Freedom is slavery
> Ignorance is strength

has nine tokens and seven types. The type-to-token ratio is often used as an elementary measure of a text's density.

6.4.2 A Crash Program to Count Words with Unix

In his famous column, *Programming Pearls*, Bentley et al. (1986) posed the following problem:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

This problem is especially interesting to us now as it is exactly the output of the first row in Table **??**.

Bentley received two solutions for it: one from Donald Knuth, the prestigious inventor of TEX, and the second in the form of a comment from Doug McIlroy, the developer of Unix pipelines. While Knuth sent an 8-page program, McIlroy proposed a compelling Unix shell script of six lines[1]. We reproduce it here (slightly modified):

1. `tr -cs 'A-Za-z' '\n' <input_file |`
   Tokenize the text in `input_file` using the Unix `tr` command. The Unix `tr` behaves like the Perl `tr` operator that we described in Sect. 3.3.4. There will be one word per line, and the output is passed to the next command.
2. `tr 'A-Z' 'a-z' |`
   Translate the uppercase characters into lowercase letters and pass the output to the next command.
3. `sort |`
   Sort the words. The identical words will be grouped together in adjacent lines.
4. `uniq -c |`
   Remove repeated lines. The identical adjacent lines will be replaced with one single line. Each unique line in the output will be preceded by the count of its duplicates in the input file (`-c`).
5. `sort -rn |`
   Sort in the reverse (`-r`) numeric (`-n`) order. The most frequent words will be sorted first.
6. `head -5`
   Print the five first lines of the file (the five most frequent words).

The two first `tr` commands do not take into account possible accented characters. To correct it, we just need to modify the character list and include accents. Nonetheless, we can apply the script as it is to English texts. On the novel *Nineteen Eighty-Four* (Orwell, 1949), the output is:

```
6518 the
3491 of
2576 a
2442 and
2348 to
```

6.4.3 Counting Words with Python

Counting words is straightforward and very fast with Python. We can obtain them with the following algorithm:

[1] Stephen Bourne, the author of the Unix Bourne shell, proposed a similar script; see Bourne (1982, pp. 196–197).

1. Tokenize the text file;
2. Count the words and store them in a dictionary;
3. Possibly, sort the words according to their alphabetical order or their frequency.

The resulting program is similar to the letter count in Sect. 2.6.8. Let us use functions to implement the different steps. For the first step, we apply a tokenizer to the text (Sect. 6.2) and we produce a list of words as output. Then, the counting function uses a dictionary with the words as keys and their frequency as value. It scans the `words` list and increments the frequency of the words as they occur.

The program reads a file and sets the characters in lower case, calls the tokenizer and the counter. It sorts the words alphabetically with `sorted(dict.keys())` and prints them with their frequency. The complete program is:

```
def tokenize(text):
    words = re.findall('\p{L}+', text)
    return words

def count_unigrams(words):
    frequency = {}
    for word in words:
        if word in frequency:
            frequency[word] += 1
        else:
            frequency[word] = 1
    return frequency

if __name__ == '__main__':
    text = sys.stdin.read().lower()
    words = tokenize(text)
    frequency = count_unigrams(words)
    for word in sorted(frequency.keys()):
        print(word, '\t', frequency[word])
```

We run it with the command:

```
python count.py < file.txt
```

If we want to sort the words by frequency, we saw in Sect. 2.6.8 that we have to assign the `key` argument the value `frequency.get` in `sorted()`. We also set the `reverse` argument to `True`:

```
sorted(frequency.keys(), key=frequency.get, reverse=True)
```

The list below shows the 10 most frequent words in *the Iliad*:

```
the      10015
and      6673
of       5643
```

```
to      3349
he      2914
his     2545
in      2252
him     1875
a       1819
you     1814
```

6.5 Retrieval and Ranking of Documents

The advent of the Web in the mid-1990s made it possible to retrieve automatically billions of documents from words or phrases they contained. Companies providing such a service became quickly among the most popular sites of the internet; Google and Bing being the most notable ones.

Web search systems or engines are based on "spiders" or "crawlers" that visit internet addresses, follow links they encounter, and collect all the pages they traverse. Crawlers can amass billions of pages every month.

6.5.1 Document Indexing

All the pages the crawlers download are tokenized and undergo a full text indexing. To carry out this first step, an indexer extracts all the words of the documents in the collection and builds a dictionary. It then links each word in the dictionary to the list of documents where this word occurs in. Such a list is called a *postings list*, where each posting in the list contains a document identifier and the word's positions in the corresponding document. The resulting data structure is called an *inverted index* and Table 6.6 shows an example of it with the two documents:

D1: Chrysler plans new investments in Latin America.
D2: Chrysler plans major investments in Mexico.

An inverted index is pretty much like a book index except that it considers all the words. When a user asks for a specific word, the search system answers with the pages that contain it. See Baeza-Yates and Ribeiro-Neto (2011) and Manning et al. (2008) for more complete descriptions.

6.5.2 Building an Inverted Index in Python

To represent the inverted index, we will use a dictionary, where the words in the collection are the keys and the postings lists, the values. In addition, we will augment the postings with the positions of the word in each document. We will also represent the posting lists as dictionaries, where the keys will be the documents identifiers and the values, the list of positions:

Table 6.6. An inverted index. Each word in the dictionary is linked to a postings list that gives all the documents in the collection where this word occurs and its positions in a document. Here, the position is the word index in the document. In the examples, a word occurs at most once in a document. This can be easily generalized to multiple occurrences

Words	Postings lists
America	(D1, 7)
Chrysler	(D1, 1) → (D2, 1)
in	(D1, 5) → (D2, 5)
investments	(D1, 4) → (D2, 4)
Latin	(D1, 6)
major	(D2, 3)
Mexico	(D2, 6)
new	(D1, 3)
plans	(D1, 2) → (D2, 2)

```
{
index[word1]: {doc_1: [pos1, pos2, ...], ..., doc_n: [pos1, ...]},
index[word2]: {doc_1: [pos1, pos2, ...], ..., doc_n: [pos1, ...]},
...
}
```

Let us first write two functions to index a single document. We extract the words with the tokenizer in Sect. 6.2.4, but instead of `findall()`, we use `finditer()` to return the match objects. We will use these match objects to extract the word positions.

```
def tokenize(text):
    """
    Uses the letters to break the text into words.
    Returns a list of match objects
    """
    words = re.finditer('\p{L}+', text)
    return words
```

Once the text is tokenized, we can build the index. We define the positions as the number of characters from the start of the file and we store them in a list:

```
def text_to_idx(words):
    """
    Builds an index from a list of match objects
    """
    word_idx = {}
    for word in words:
        try:
            word_idx[word.group()].append(word.start())
```

```
        except:
            word_idx[word.group()] = [word.start()]
    return word_idx
```

Using these two functions, we can index documents, for instance *A Tale of Two Cities* by Dickens[2]:

```
> text = open('A Tale of Two Cities.txt').read().lower().strip()
> index = text_to_idx(tokenize(text))
```

and extract the word positions from the index:

```
> index['congratulate']
[28076, 661716]
> index['deserve']
[269196, 618140, 669252]
> index['vendor']
[218582, 218631, 219234, 635168]
```

We finally build the index of all books in the collection with a loop over the list of files:

```
master_index = {}
for file in corpus_files:
    text = open(file).read().lower().strip()
    words = tokenize(text)
    idx = text_to_idx(words)
    for word in idx:
        if word in master_index:
            master_index[word][file] = idx[word]
        else:
            master_index[word] = {}
            master_index[word][file] = idx[word]
```

Applying this program to a collection of works by Dickens results in a master index from which we can find all the positions of a word, such as *vendor*, in the documents that contain it:

```
> master_index['vendor'] =
{'Dombey and Son.txt': [1080291],
'A Tale of Two Cities.txt': [218582, 218631, 219234, 635168],
'The Pickwick Papers.txt': [28715],
'Bleak House.txt': [1474429],
'Oliver Twist.txt': [788457]}
```

[2] Retrieved from the Gutenberg Project, `www.gutenberg.org`, November 3, 2016.

Table 6.7. The vectors representing the two documents in Sect. 6.5.1. The words have been normalized in lowercase letters

D#\ Words	america	chrysler	in	investments	latin	major	mexico	new	plans
1	1	1	1	1	1	0	0	1	1
2	0	1	1	1	0	1	1	0	1

Table 6.8. The word by document matrix. Each cell (w_i, D_j) contains the frequency of w_i in document D_j

D#\Words	w_1	w_2	w_3	...	w_m
D_1	$C(w_1,D_1)$	$C(w_2,D_1)$	$C(w_3,D_1)$	...	$C(w_m,D_1)$
D_2	$C(w_1,D_2)$	$C(w_2,D_2)$	$C(w_3,D_2)$	...	$C(w_m,D_2)$
...					
D_n	$C(w_1,D_n)$	$C(w_2,D_n)$	$C(w_3,D_n)$	...	$C(w_m,D_n)$

6.5.3 Representing Documents as Vectors

Once indexed, search engines compare, categorize, and rank documents using statistical or popularity models. The vector space model (Salton, 1988) is a widely used representation to carry this out. The idea is to represent the documents in a vector space whose axes are the words. Documents are then vectors in a space of words. As the word order plays no role in the representation, it is often called a *bag-of-word model*.

Let us first suppose that the document coordinates are the occurrence counts of each word. A document would be represented as: $\mathbf{d} = (C(w_1), C(w_2), C(w_3), ..., C(w_n))$. Table 6.7 shows the document vectors representing the examples in Sect. 6.5.1, and Table 6.8 shows a general matrix representing a collection of documents, where each cell (w_i, D_j) contains the frequency of w_i in document D_j.

Using the vector space model, we can measure the similarity between two documents by the angle they form in the vector space. It is easier to compute the cosine of the angle, which is formulated as:

$$\cos(\mathbf{q}, \mathbf{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \sqrt{\sum_{i=1}^{n} d_i^2}}.$$

6.5.4 Vector Coordinates

In fact, most of the time, the rough word counts that are used as coordinates in the vectors are replaced by a more elaborate term: the term frequency times the inverse document frequency, better known as $tf \times idf$ (Salton, 1988). To examine how it works, let us take the phrase *internet in Somalia* as an example.

A document that contains many *internet* words is probably more relevant than a document that has only one. The frequency of a term i in a document j reflects this. It is a kind of a "mass" relevance. For each vector, the term frequencies $tf_{i,j}$ are often normalized by the sum of the frequencies of all the terms in the document and defined as:

$$tf_{i,j} = \frac{t_{i,j}}{\sum_i t_{i,j}},$$

or as the Euclidean norm:

$$tf_{i,j} = \frac{t_{i,j}}{\sqrt{\sum_i t_{i,j}^2}},$$

where $t_{i,j}$ is the frequency of term i in document j – the number of occurrences of term i in document j.

Instead of a sum, we can also use the maximum count over all the terms as normalization factor. The term frequency of the term i in document j is then defined as:

$$tf_{i,j} = \frac{t_{i,j}}{\max_i t_{i,j}}.$$

However, since *internet* is a very common word, it is not specific. The number of documents that contain it must downplay its importance. This is the role of the inverse document frequency (Spärck Jones, 1972):

$$idf_i = \log(\frac{N}{n_i}),$$

where N is the total number of documents in the collection – the total number of pages the crawler has collected – divided by the number of pages n_i, where a term i occurs at least once. *Somalia* probably appears in fewer documents than *internet* and idf_i will give it a chance. The weight of a term i in document j is finally defined as

$$tf_{i,j} \times \log(\frac{N}{n_i}).$$

In this section, we gave one definition of $tf \times idf$. In fact, this formula can vary depending on the application. Salton and Buckley (1987) reported 287 variants of it and compared their respective merits. BM25 and BM25F (Zaragoza et al., 2004) are extensions of $tf \times idf$ that take into account the document length.

6.5.5 Ranking Documents

The user may query a search engine with a couple of words or a phrase. Most systems will then answer with the pages that contain all the words and any of the words of the question. Some questions return hundreds or even thousands of valid documents. Ranking a document consists in projecting the space to that of the question words

using the cosine. With this model, higher cosines will indicate better relevance. In addition to $tf \times idf$, search systems may employ heuristics such as giving more weight to the words in the title of a page (Mauldin and Leavitt, 1994).

Google's PageRank algorithm (Brin and Page, 1998) uses a different technique that takes into account the page popularity. PageRank considers the "backlinks", the links pointing to a page. The idea is that a page with many backlinks is likely to be a page of interest. Each backlink has a specific weight, which corresponds to the rank of the page it comes from. The page rank is simply defined as the sum of the ranks of all its backlinks. The importance of a page is spread through its forward links and contributes to the popularity of the pages it points to. The weight of each of these forward links is the page rank divided by the count of the outgoing links. The ranks are propagated in a document collection until they converge.

6.5.6 Categorizing Text

Text categorization (or classification) is a task related to ranking, but instead of associating documents to queries, we assign one or more classes to a text. The text size can range from a few words to entire books. In sentiment analysis (or opinion mining), the goal is to classify judgments or emotions expressed, for instance, in product reviews collected from consumer forums, into three base categories: positive, negative, or neutral; in spam detection, the categorizer classifies electronic messages into two classes: *spam* or *no spam*.

The Reuters corpus of newswire articles provides another example of a text collection that also serves as a standardized benchmark for categorization algorithms (Lewis et al., 2004). This corpus consists of 800,000 economic newswires in English and about 500,000 in 13 other languages, where each newswire is manually annotated with one or more topics selected from a set of 103 predefined categories, such as:

C11: STRATEGY/PLANS,
C12: LEGAL/JUDICIAL,
C13: REGULATION/POLICY,
C14: SHARE LISTINGS
etc.

Using manually-categorized corpora, like the Reuters corpus, and the vector space model, we can apply supervised machine-learning techniques to train classifiers (see Sect. 5.4). The training procedure uses a bag-of-word representation of the documents, either with Boolean features, term frequencies, or $tf \times idf$, and their classes as input. Support vector machines and logistic regression are two efficient techniques to carry out text classification. Joachims (2002) describes a state-of-the-art classifier based on support vector machines, while LibShortText (Yu et al., 2013) is an open source library consisting of support vector machine and logistic regression algorithms, and different types of preprocessing and feature representations.

6.6 Further Reading

There are several toolkits available from the Internet to carry out tokenization, sentence detection, and language modeling:

1. Apache OpenNLP is a complete suite of logistic regression-based modules that includes, *inter alia*, a sentence detector, a tokenizer, and a document categorizer (`http://opennlp.apache.org/`).
2. The SRI Language Modeling collection (Stolcke, 2002) is a C++ package to create and experiment with language models (`http://www.speech.sri.com/`).
3. The CMU-Cambridge Statistical Language Modeling Toolkit (Clarkson and Rosenfeld, 1997) is another set of tools (`http://svr-www.eng.cam.ac.uk/~prc14/toolkit.html`).

Retrieval and ranking of documents have experienced a phenomenal growth since the beginning of the Web, making search sites the most popular services of the Internet. For complete reviews of techniques on information retrieval, see Manning et al. (2008) or Baeza-Yates and Ribeiro-Neto (2011).

Lucene is a popular open-source library for information retrieval. It is used in scores of web sites such as Twitter and Wikipedia to carry out document indexing and search (`http://lucene.apache.org/`).

Exercises

6.1. Write a sentence detector and a tokenizer using logistic regression.

6.2. Retrieve a text you like on the Internet. Give the five most frequent words.

6.3. Retrieve a collection of documents and index them all using an inverted index. Compute for each word and each document its $tf \cdot idf$ value. Represent the documents as bags of words with $tf \cdot idf$ as coordinates. Compare the documents pairwise using the cosine similarity and tell what are the two most similar documents of your collection.

7

Word Sequences

> On trouve ainsi qu'un événement étant arrivé de suite, un nombre quelconque de fois, la probabilité qu'il arrivera encore la fois suivante, est égale à ce nombre augmenté de l'unité, divisé par le même nombre augmenté de deux unités. En faisant, par exemple, remonter la plus ancienne époque de l'histoire, à cinq mille ans, ou à 1826213 jours, et le Soleil s'étant levé constamment, dans cet intervalle, à chaque révolution de vingt-quatre heures, il y a 1826214 à parier contre un qu'il se lèvera encore demain.
>
> Pierre-Simon Laplace. *Essai philosophique sur les probabilités*. 1840.
> See explanations in Sect. 7.4.2.

7.1 Modeling Word Sequences

We saw in Chap. 3 that words have specific contexts of use. Pairs of words like *strong* and *tea* or *powerful* and *computer* are not random associations but the result of a preference. A native speaker will use them naturally, while a learner will have to learn them from books – dictionaries – where they are explicitly listed. Similarly, the words *rider* and *writer* sound much alike in American English, but they are likely to occur with different surrounding words. Hence, hearing an ambiguous phonetic sequence, a listener will discard the improbable *rider of books* or *writer of horses* and prefer *writer of books* or *rider of horses* (Church and Mercer, 1993).

In lexicography, extracting recurrent pairs of words – collocations – is critical to finding the possible contexts of a word and citing real examples of its use. In speech recognition, the statistical estimate of a word sequence – also called a **language model** – is a key part of the recognition process. The language model component of a speech recognition system enables the system to predict the next word given a sequence of previous words: *the writer of books, novels, poetry,* etc., rather than of *the writer of hooks, nobles, poultry*.

Knowing the frequency of words and sequences of words is crucial in many fields of language processing. In addition to speech recognition and lexicography, they

Table 7.1. Ranking and generating words using trigrams. After Jelinek (1990)

Word	Rank	More likely alternatives
We	9	*The This One Two A Three Please In*
need	7	*are will the would also do*
to	1	
resolve	85	*have know do...*
all	9	*the this these problems...*
of	2	*the*
the	1	
important	657	*document question first...*
issues	14	*thing point to...*
within	74	*to of and in that...*
the	1	
next	2	*company*
two	5	*page exhibit meeting day*
days	5	*weeks years pages months*

include parsing, semantic interpretation, and translation. In this chapter, we introduce techniques to obtain word frequencies from a corpus and to build language models. We also describe a set of related concepts that are essential to understand them.

7.2 *N*-grams

Collocations and language models use the frequency of pairs of adjacent words: **bigrams**, for example, how many *of the* there are in this text; of word triples: **trigrams**; and more generally of fixed sequences of n words: ***n*-grams**. In lexical statistics, single words are called **unigrams**.

Jelinek (1990) exemplified corpus statistics and trigrams with the sentence

We need to resolve all of the important issues within the next two days

selected from a 90-million-word corpus of IBM office correspondences. Table 7.1 shows each word of this sentence, its rank in the corpus, and other words ranking before it according to a linear combination of trigram, bigram, and unigram probabilities. In this corpus, *We* is the ninth most probable word to begin a sentence. More likely words are *The*, *This*, etc. Following *We*, *need* is the seventh most probable word. More likely bigrams are *We are*, *We will*, *We the*, *We would*.... Knowing that the words *We need* have been written, *to* is the most likely word to come after them. Similarly, *the* is the most probable word to follow *all of*.

7.2.1 Counting Bigrams with Unix

In addition, it is easy to extend the counts to bigrams. We need first to create a file, where each line contains a bigram: the words at index i and $i+1$ on the same line separated with a blank. We use the Unix commands:

1. `tr -cs 'A-Za-z' '\n' < input_file > token_file`
   Tokenize the input and create a file with the unigrams.
2. `tail +2 < token_file > next_token_file`
   Create a second unigram file starting at the second word of the first tokenized file (`+2`).
3. `paste token_file next_token_file > bigrams`
   Merge the lines (the tokens) pairwise. Each line of `bigrams` contains the words at index i and $i+1$ separated with a tabulation.
4. And we count the bigrams as in the previous script.

7.2.2 Counting Bigrams with Python

We count bigrams just as we did with unigrams. The only difference is that we use pairs of adjacent words instead of words. We extract these pairs, (w_i, w_{i+1}), with the slice notation: `words[i:i+2]` that produces a list of two strings. As with unigrams, we use the bigrams as the keys of a dictionary and their frequencies as the values. We need then to make sure that we have the right data type for this. Python dictionaries only accept immutable structures as keys (see Sect. 2.6.6) and we hence convert our bigrams to tuples. We create the bigram list with a list comprehension:

```
bigrams = [tuple(words[idx:idx + 2])
           for idx in range(len(words) - 1)]
```

The rest of the `count_bigrams` function is nearly identical to `count_unigrams`. As input, it uses the same list of words:

```
def count_bigrams(words):
    bigrams = [tuple(words[idx:idx + 2])
               for idx in range(len(words) - 1)]
    frequencies = {}
    for bigram in bigrams:
        if bigram in frequencies:
            frequencies[bigram] += 1
        else:
            frequencies[bigram] = 1
    return frequencies
```

The five most frequent bigrams of the *the Iliad* are:

```
('of', 'the')    1247
('son', 'of')    826
('to', 'the')    584
('and', 'the')   525
('in', 'the')    505
```

The bigram count can easily be generalized to *n*-grams with the function:

```
def count_ngrams(words, n):
    ngrams = [tuple(words[idx:idx + n])
              for idx in range(len(words) - n + 1)]
    frequencies = {}
    for ngram in ngrams:
        if ngram in frequencies:
            frequencies[ngram] += 1
        else:
            frequencies[ngram] = 1
    return frequencies
```

Similarly, we obtain the five most frequent trigrams from the *the Iliad*:

```
('the', 'son', 'of')     355
('of', 'the', 'achaeans')        203
('son', 'of', 'atreus')          119
('son', 'of', 'peleus')          98
('the', 'trojans', 'and')        96
```

7.3 Probabilistic Models of a Word Sequence

7.3.1 The Maximum Likelihood Estimation

We observed in Table 7.1 that some word sequences are more likely than others. Using a statistical model, we can quantify these observations. The model will enable us to assign a probability to a word sequence as well as to predict the next word to follow the sequence.

Let $S = w_1, w_2, ..., w_i, ..., w_n$ be a word sequence. Given a training corpus, an intuitive estimate of the probability of the sequence, $P(S)$, is the relative frequency of the string $w_1, w_2, ..., w_i, ..., w_n$ in the corpus. This estimate is called the *maximum likelihood estimate* (MLE):

$$P_{\text{MLE}}(S) = \frac{C(w_1, ..., w_n)}{N},$$

where $C(w_1, ..., w_n)$ is the frequency or count of the string $w_1, w_2, ..., w_i, ..., w_n$ in the corpus, and N is the total number of strings of length n.

Most of the time, however, it is impossible to obtain this estimate. Even when corpora reach billions of words, they have a limited size, and it is unlikely that we can always find the exact sequence we are searching. We can try to simplify the computation and decompose $P(S)$ a step further using the chain rule as:

$$\begin{aligned} P(S) &= P(w_1, ..., w_n), \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)...P(w_n|w_1, ..., w_{n-1}), \\ &= \prod_{i=1}^{n} P(w_i|w_1, ..., w_{i-1}). \end{aligned}$$

The probability $P(It, was, a, bright, cold, day, in, April)$ from *Nineteen Eighty-Four* by George Orwell corresponds then to the probability of having *It* to begin the sentence, then *was* knowing that we have *It* before, then *a* knowing that we have *It was* before, and so on, until the end of the sentence. It yields the product of conditional probabilities:

$$P(S) = P(It) \times P(was|It) \times P(a|It, was) \times P(bright|It, was, a) \times ... \\ \times P(April|It, was, a, bright, ..., in).$$

To estimate $P(S)$, we need to know unigram, bigram, trigram, so far, so good, but also 4-gram, 5-gram, and even 8-gram statistics. Of course, no corpus is big enough to produce them. A practical solution is then to limit the n-gram length to 2 or 3, and thus to approximate them to bigrams:

$$P(w_i|w_1, w_2, ..., w_{i-1}) \approx P(w_i|w_{i-1}),$$

or trigrams:

$$P(w_i|w_1, w_2, ..., w_{i-1}) \approx P(w_i|w_{i-2}, w_{i-1}).$$

Using a trigram language model, $P(S)$ is approximated as:

$$P(S) \approx P(It) \times P(was|It) \times P(a|It, was) \times P(bright|was, a) \times ... \\ \times P(April|day, in).$$

Using a bigram grammar, the general case of a sentence probability is:

$$P(S) \approx P(w_1) \prod_{i=2}^{n} P(w_i|w_{i-1}),$$

with the estimate

$$P_{\text{MLE}}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{\sum_w C(w_{i-1}, w)} = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}.$$

Similarly, the trigram maximum likelihood estimate is:

$$P_{\text{MLE}}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}.$$

And the general case of n-gram estimation is:

$$P_{\text{MLE}}(w_{i+n}|w_{i+1}, ..., w_{i+n-1}) = \frac{C(w_{i+1}, ..., w_{i+n})}{\sum_w C(w_{i+1}, ..., w_{i+n-1}, w)}, \\ = \frac{C(w_{i+1}, ..., w_{i+n})}{C(w_{i+1}, ..., w_{i+n-1})}.$$

As the probabilities we obtain are usually very low, it is safer to represent them as a sum of logarithms in practical applications. We will then use:

$$\log P(S) \approx \log P(w_1) + \sum_{i=2}^{n} \log P(w_i|w_{i-1}),$$

instead of $P(S)$. Nonetheless, in the following sections, as our example corpus is very small, we will compute the probabilities using products.

7.3.2 Using ML Estimates with *Nineteen Eighty-Four*

Training and Testing the Language Model.

Before computing the probability of a word sequence, we must train the language model. The corpus used to derive the n-gram frequencies is classically called the **training set**, and the corpus on which we apply the model, the **test set**. Both sets should be distinct. If we apply a language model to a word sequence, which is part of the training corpus, its probability will be biased to a higher value, and thus will be inaccurate. The training and test sets can be balanced or not, depending on whether we want them to be specific of a task or more general.

For some models, we need to optimize parameters in order to obtain the best results. Again, it would bias the results if at the same time, we carry out the optimization on the test set and run the evaluation on it. For this reason some models need a separate **development set** to fine-tune their parameters.

In some cases, especially with small corpora, a specific division between training and test sets may have a strong influence on the results. It is then preferable to apply the training and testing procedure several times with different sets and average the results. The method is to randomly divide the corpus into two sets. We learn the parameters from the training set, apply the model to the test set, and repeat the process with a new random division, for instance, ten times. This method is called **cross-validation**, or ten-fold cross-validation if we repeat it ten times. Cross-validation smoothes the impact of a specific partition of the corpus.

Marking up the Corpus.

Most corpora use some sort of markup language. The most common markers of N-gram models are the sentence delimiters `<s>` to mark the start of a sentence and `</s>` at its end. For example:

`<s>` *It was a bright cold day in April* `</s>`

Depending on the application, both symbols can be counted in the n-gram frequencies just as the other tokens or can be considered as context cues. Context cues are vocabulary items that appear in the condition part of the probability but are never predicted – they never occur in the right part. In many models, `<s>` is a context cue and `</s>` is part of the vocabulary. We will adopt this convention in the next examples.

The Vocabulary.

We have defined language models that use a predetermined and finite set of words. This is never the case in reality, and the models will have to handle out-of-vocabulary (OOV) words. Training corpora are typically of millions, or even billions, of words. However, whatever the size of a corpus, it will never have a complete coverage of the

vocabulary. Some words that are unseen in the training corpus are likely to occur in the test set. In addition, frequencies of rare words will not be reliable.

There are two main types of methods to deal with OOV words:

- The first method assumes a **closed vocabulary**. All the words both in the training and the test sets are known in advance. Depending on the language model settings, any word outside the vocabulary will be discarded or cause an error. This method is used in some applications, like voice control of devices.
- The **open vocabulary** makes provisions for new words to occur with a specific symbol, `<UNK>`, called the unknown token. All the OOV words are mapped to `<UNK>`, both in the training and test sets.

The vocabulary itself can come from an external dictionary. It can also be extracted directly from the training set. In this case, it is common to exclude the rare words, notably those seen only once – the *hapax legomena*. The vocabulary will then consist of the most frequent types of the corpus, for example, the 20,000 most frequent types. The other words, unseen or with a frequency lower than a cutoff value, 1, 2, or up to 5, will be mapped to `<UNK>`.

Computing a Sentence Probability.

We trained a bigram language model on a very small corpus consisting of the three chapters of *Nineteen Eighty-Four*. We kept the appendix, "The Principles of Newspeak," as the test set and we selected this sentence from it:

> `<s>` *A good deal of the literature of the past was, indeed, already being transformed in this way* `</s>`

We first normalized the text: we created a file with one sentence per line. We inserted automatically the delimiters `<s>` and `</s>`. We removed the punctuation, parentheses, quotes, stars, dashes, tabulations, and double white spaces. We set all the words in lowercase letters. We counted the words, and we produced a file with the unigram and bigram counts.

The training corpus has 115,212 words; 8635 types, including 3928 hapax legomena; and 49,524 bigrams, where 37,365 bigrams have a frequency of 1. Table 7.2 shows the unigram and bigram frequencies for the words of the test sentence.

All the words of the sentence have been seen in the training corpus, and we can compute a probability estimate of it using the unigram relative frequencies:

$$\begin{aligned} P(S) &\approx P(a) \times P(good) \times ... \times P(way) \times P(</\mathtt{s}>), \\ &\approx 3.67 \times 10^{-48}. \end{aligned}$$

As $P(<\mathtt{s}>)$ is a constant that would scale all the sentences by the same factor, whether we use unigrams or bigrams, we excluded it from the $P(S)$ computation.

The bigram estimate is defined as:

$$P(S) \approx P(a|<\mathtt{s}>) \times P(good|a) \times ... \times P(way|this) \times P(</\mathtt{s}>|way).$$

Table 7.2. Frequencies of unigrams and bigrams. We excluded the <s> symbols from the word counts

w_i	$C(w_i)$	#words	$P_{\text{MLE}}(w_i)$	w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{MLE}}(w_i \mid w_{i-1})$
<s>	7072	–	–	–	–		
a	2482	108140	0.023	<s> *a*	133	7072	0.019
good	53	108140	0.00049	*a good*	14	2482	0.006
deal	5	108140	4.62×10^{-5}	*good deal*	0	53	0.0
of	3310	108140	0.031	*deal of*	1	5	0.2
the	6248	108140	0.058	*of the*	742	3310	0.224
literature	7	108140	6.47×10^{-5}	*the literature*	1	6248	0.00016
of	3310	108140	0.031	*literature of*	3	7	0.429
the	6248	108140	0.058	*of the*	742	3310	0.224
past	99	108140	0.00092	*the past*	70	6248	0.011
was	2211	108140	0.020	*past was*	4	99	0.040
indeed	17	108140	0.00016	*was indeed*	0	2211	0.0
already	64	108140	0.00059	*indeed already*	0	17	0.0
being	80	108140	0.00074	*already being*	0	64	0.0
transformed	1	108140	9.25×10^{-6}	*being transformed*	0	80	0.0
in	1759	108140	0.016	*transformed in*	0	1	0.0
this	264	108140	0.0024	*in this*	14	1759	0.008
way	122	108140	0.0011	*this way*	3	264	0.011
</s>	7072	108140	0.065	*way* </s>	18	122	0.148

and has a zero probability. This is due to **sparse data**: the fact that the corpus is not big enough to have all the bigrams covered with a realistic estimate. We shall see in the next section how to handle them.

7.4 Smoothing N-gram Probabilities

7.4.1 Sparse Data

The approach using the maximum likelihood estimation has an obvious disadvantage because of the unavoidably limited size of the training corpora. Given a vocabulary of 20,000 types, the potential number of bigrams is $20,000^2 = 400,000,000$, and with trigrams, it amounts to the astronomic figure of $20,000^3 = 8,000,000,000,000$. No corpus yet has the size to cover the corresponding word combinations.

Among the set of potential n-grams, some are almost impossible, except as random sequences generated by machines; others are simply unseen in the corpus. This phenomenon is referred to as **sparse data**, and the maximum likelihood estimator gives no hint on how to estimate their probability.

In this section, we introduce **smoothing** techniques to estimate probabilities of unseen n-grams. As the sum of probabilities of all the n-grams of a given length is

1, smoothing techniques also have to rearrange the probabilities of the observed n-grams. Smoothing allocates a part of the probability mass to the unseen n-grams that, as a counterpart, it shifts – or **discounts** – from the other n-grams.

7.4.2 Laplace's Rule

Laplace's rule (Laplace, 1820, p. 17) is probably the oldest published method to cope with sparse data. It just consists in adding one to all the counts. For this reason, some authors also call it the add-one method.

Laplace wanted to estimate the probability of the sun to rise tomorrow and he imagined this rule: he set both event counts, rise and not rise, arbitrarily to one, and he incremented them with the corresponding observations. From the beginning of time, humans had seen the sun rise every day. Laplace derived the frequency of this event from what he believed to be the oldest epoch of history: five thousand years or 1,826,213 days. As nobody observed the sun not rising, he obtained the chance for the sun to rise tomorrow of 1,826,214 to 1.

Laplace's rule states that the frequency of unseen n-grams is equal to 1 and the general estimate of a bigram probability is:

$$P_{\text{Laplace}}(w_i|w_{i-1}) = \frac{C(w_{i-1},w_i)+1}{\sum_w (C(w_{i-1},w)+1)} = \frac{C(w_{i-1},w_i)+1}{C(w_{i-1})+Card(V)},$$

where $Card(V)$ is the number of word types. The denominator correction is necessary to have the probability sum equal to 1.

With Laplace's rule, we can use bigrams to compute the sentence probability (Table 7.3):

$$\begin{aligned} P_{\text{Laplace}}(S) &\approx P(a|\texttt{<s>}) \times P(good|a) \times ... \times P(\texttt{</s>}|way), \\ &\approx 4.62 \times 10^{-57}. \end{aligned}$$

Laplace's method is easy to understand and implement. It has an obvious drawback however: it shifts an enormous mass of probabilities to the unseen n-grams and gives them a considerable importance. The frequency of the unlikely bigram *the of* will be 1, a quarter of the much more common *this way*.

The **discount** value is the ratio between the smoothed frequencies and their actual counts in the corpus. The bigram *this way* has been discounted by $0.011/0.00045 = 24.4$ to make place for the unseen bigrams. This is unrealistic and shows the major drawback of this method. For this small corpus, Laplace's rule applied to bigrams has a result opposite to what we wished. It has not improved the sentence probability over the unigrams. This would mean that a bigram language model is worse than words occurring randomly in the sentence.

If adding 1 is too much, why not try less, for instance, 0.5? This is the idea of Lidstone's rule. This value is denoted λ. The new formula is then:

$$P_{\text{Lidstone}}(w_i|w_{i-1}) = \frac{C(w_{i-1},w_i)+\lambda}{C(w_{i-1})+\lambda Card(V)},$$

which, however, is not a big improvement.

Table 7.3. Frequencies of bigrams using Laplace's rule

w_{i-1}, w_i	$C(w_{i-1}, w_i)+1$	$C(w_{i-1})+Card(V)$	$P_{\text{Lap}}(w_i \mid w_{i-1})$
`<s>` *a*	133 + 1	7072 + 8635	0.0085
a good	14 + 1	2482 + 8635	0.0013
good deal	0 + 1	53 + 8635	0.00012
deal of	1 + 1	5 + 8635	0.00023
of the	742 + 1	3310 + 8635	0.062
the literature	1 + 1	6248 + 8635	0.00013
literature of	3 + 1	7 + 8635	0.00046
of the	742 + 1	3310 + 8635	0.062
the past	70 + 1	6248 + 8635	0.0048
past was	4 + 1	99 + 8635	0.00057
was indeed	0 + 1	2211 + 8635	0.000092
indeed already	0 + 1	17 + 8635	0.00012
already being	0 + 1	64 + 8635	0.00011
being transformed	0 + 1	80 + 8635	0.00011
transformed in	0 + 1	1 + 8635	0.00012
in this	14 + 1	1759 + 8635	0.0014
this way	3 + 1	264 + 8635	0.00045
way `</s>`	18 + 1	122 + 8635	0.0022

7.4.3 Good–Turing Estimation

The Good–Turing estimation (Good, 1953) is one of the most efficient smoothing methods. As with Laplace's rule, it reestimates the counts of the n-grams observed in the corpus by discounting them, and it shifts the probability mass it has shaved to the unseen bigrams. The discount factor is variable, however, and depends on the number of times a n-gram has occurred in the corpus. There will be a specific discount value to n-grams seen once, another one to bigrams seen twice, a third one to those seen three times, and so on.

Let us denote N_c the number of n-grams that occurred exactly c times in the corpus. N_0 is the number of unseen n-grams, N_1 the number of n-grams seen once, N_2 the number of n-grams seen twice, and so on. If we consider bigrams, the value N_0 is $Card(V)^2$ minus all the bigrams we have seen.

The Good–Turing method reestimates the frequency of n-grams occurring c times using the formula:

$$c^* = (c+1)\frac{E(N_{c+1})}{E(N_c)},$$

where $E(x)$ denotes the expectation of the random variable x. This formula is usually approximated as:

$$c^* = (c+1)\frac{N_{c+1}}{N_c}.$$

To understand how this formula was designed, let us take the example of the unseen bigrams with $c = 0$. Let us suppose that we draw a sequence of bigrams to

build our training corpus, and the last bigram we have drawn was unseen before. From this moment, there is one occurrence of it in the training corpus and the count of bigrams in the same case is N_1. Using the maximum likelihood estimation, the probability to draw such an unseen bigram is then the count of bigrams seen once divided by the total count of the bigrams seen so far: N_1/N. We obtain the probability to draw one specific unseen bigram by dividing this term by the count of unseen bigrams:

$$\frac{1}{N} \times \frac{N_1}{N_0}.$$

Hence, the Good–Turing reestimated count of an unseen n-gram is $c^* = \frac{N_1}{N_0}$. Similarly, we would have $c^* = \frac{2N_2}{N_1}$ for an n-gram seen once in the training corpus.

The three chapters in *Nineteen Eighty-Four* contain 37,365 unique bigrams and 5820 bigrams seen twice. Its vocabulary of 8635 words generates $8635^2 =$ 74,563,225 bigrams, of which 74,513,701 are unseen. The Good–Turing method reestimates the frequency of each unseen bigram to $37,365/74,513,701 = 0.0005$, and unique bigrams to $2 \times (5,820/37,365) = 0.31$. Table 7.4 shows the complete the reestimated frequencies for the n-grams up to 9.

In practice, only high values of N_c are reliable, which correspond to low values of c. In addition, above a certain threshold, most frequencies of frequency will be equal to zero. Therefore, the Good–Turing estimation is applied for $c < k$, where k is a constant set to 5, 6, ..., or 10. Other counts are not reestimated. See Katz (1987) for the details.

The probability of a n-gram is given by the formula:

$$P_{\mathrm{GT}}(w_1,...,w_n) = \frac{c^*(w_1,...,w_n)}{N},$$

where c^* is the reestimated count of $w_1...w_n$, and N the original count of n-grams in the corpus. The conditional frequency is

$$P_{\mathrm{GT}}(w_n|w_1,...,w_{n-1}) = \frac{c^*(w_1,...,w_n)}{C(w_1,...,w_{n-1})}.$$

Table 7.5 shows the conditional probabilities, where only frequencies less than 10 have been reestimated. The sentence probability using bigrams is 2.56×10^{-50}. This is better than with Laplace's rule, but as the corpus is very small, still greater than the unigram probability.

7.5 Using N-grams of Variable Length

In the previous section, we used smoothing techniques to reestimate the probability of n-grams of constant length, whether they occurred in the training corpus or not. A property of these techniques is that they assign the same probability to all the unseen n-grams.

Table 7.4. The reestimated frequencies of the bigrams

Frequency of occurrence	N_c	c^*
0	74,513,701	0.0005
1	37,365	0.31
2	5,820	1.09
3	2,111	2.02
4	1,067	3.37
5	719	3.91
6	468	4.94
7	330	6.06
8	250	6.44
9	179	8.94

Table 7.5. The conditional frequencies using the Good–Turing method. We have not reestimated the frequencies when they are greater than 9

w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$c^*(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\mathrm{GT}}(w_i \mid w_{i-1})$
`<s>` *a*	133	133	7072	0.019
a good	14	14	2482	0.006
good deal	0	0.0005	53	9.46×10^{-6}
deal of	1	0.31	5	0.062
of the	742	742	3310	0.224
the literature	1	0.31	6248	4.99×10^{-5}
literature of	3	2.02	7	0.29
of the	742	742	3310	0.224
the past	70	70	6248	0.011
past was	4	3.37	99	0.034
was indeed	0	0.0005	2211	2.27×10^{-7}
indeed already	0	0.0005	17	2.95×10^{-5}
already being	0	0.0005	64	7.84×10^{-6}
being transformed	0	0.0005	80	6.27×10^{-6}
transformed in	0	0.0005	1	0.00050
in this	14	14	1759	0.008
this way	3	2.02	264	0.0077
way `</s>`	18	18	122	0.148

Another strategy is to rely on the frequency of observed sequences but of lesser length: $n-1$, $n-2$, and so on. As opposed to smoothing, the estimate of each unseen n-gram will be specific to the words it contains. In this section, we introduce two techniques: the linear interpolation and Katz's back-off model.

7.5.1 Linear Interpolation

Linear interpolation, also called deleted interpolation (Jelinek and Mercer, 1980), combines linearly the maximum likelihood estimates from length 1 to n. For tri-

Table 7.6. Interpolated probabilities of bigrams using the formula $\lambda_2 P_{\text{MLE}}(w_i|w_{i-1}) + \lambda_1 P_{\text{MLE}}(w_i)$, $\lambda_2 = 0.7$, and $\lambda_1 = 0.3$. The total number of words is 108,140

w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{MLE}}(w_i\|w_{i-1})$	$P_{\text{MLE}}(w_i)$	$P_{\text{Interp}}(w_i\|w_{i-1})$
`<s>` *a*	133	7072	0.019	0.023	0.020
a good	14	2482	0.006	0.00049	0.0041
good deal	0	53	0.0	4.62×10^{-5}	1.38×10^{-5}
deal of	1	5	0.2	0.031	0.149
of the	742	3310	0.224	0.058	0.174
the literature	1	6248	0.00016	6.47×10^{-5}	0.000131
literature of	3	7	0.429	0.031	0.309
of the	742	3310	0.224	0.058	0.174
the past	70	6248	0.011	0.00092	0.00812
past was	4	99	0.040	0.020	0.0344
was indeed	0	2211	0.0	0.00016	4.71×10^{-5}
indeed already	0	17	0.0	0.00059	0.000177
already being	0	64	0.0	0.00074	0.000222
being transformed	0	80	0.0	9.25×10^{-6}	2.77×10^{-6}
transformed in	0	1	0.0	0.016	0.00488
in this	14	1759	0.008	0.0024	0.0063
this way	3	264	0.011	0.0011	0.00829
way `</s>`	18	122	0.148	0.065	0.123

grams, it corresponds to:

$$P_{\text{Interpolation}}(w_n|w_{n-2}, w_{n-1}) = \lambda_3 P_{\text{MLE}}(w_n|w_{n-2}, w_{n-1}) + \lambda_2 P_{\text{MLE}}(w_n|w_{n-1}) + \lambda_1 P_{\text{MLE}}(w_n),$$

where $0 \leq \lambda_i \leq 1$ and $\sum_{i=1}^{3} \lambda_i = 1$.

The values can be constant and set by hand, for instance, $\lambda_3 = 0.6$, $\lambda_2 = 0.3$, and $\lambda_1 = 0.1$. They can also be trained and optimized from a corpus (Jelinek, 1997).

Table 7.6 shows the interpolated probabilities of bigrams with $\lambda_2 = 0.7$ and $\lambda_1 = 0.3$. The sentence probability using these interpolations is 9.46×10^{-45}.

We can now understand why bigram *we the* is ranked so high in Table 7.1 after *we are* and *we will*. Although it can occur in English, as in the American constitution, *We the people*..., it is not a very frequent combination. In fact, the estimation has been obtained with an interpolation where the term $\lambda_1 P_{\text{MLE}}(the)$ boosted the bigram to the top because of the high frequency of *the*.

7.5.2 Back-off

The idea of the back-off model is to use the frequency of the longest available n-grams, and if no n-gram is available to back off to the $(n-1)$-grams, and then to $(n-2)$-grams, and so on. If n equals 3, we first try trigrams, then bigrams, and finally

Table 7.7. Probability estimates using an elementary backoff technique

w_{i-1}, w_i	$C(w_{i-1}, w_i)$		$C(w_i)$	$P_{\text{Backoff}}(w_i \mid w_{i-1})$
`<s>`			7072	—
`<s>` *a*	133		2482	0.019
a good	14		53	0.006
good deal	0	backoff	5	4.62×10^{-5}
deal of	1		3310	0.2
of the	742		6248	0.224
the literature	1		7	0.00016
literature of	3		3310	0.429
of the	742		6248	0.224
the past	70		99	0.011
past was	4		2211	0.040
was indeed	0	backoff	17	0.00016
indeed already	0	backoff	64	0.00059
already being	0	backoff	80	0.00074
being transformed	0	backoff	1	9.25×10^{-6}
transformed in	0	backoff	1759	0.016
in this	14		264	0.008
this way	3		122	0.011
way `</s>`	18		7072	0.148

unigrams. For a bigram language model, the back-off probability can be expressed as:

$$P_{\text{Backoff}}(w_i \mid w_{i-1}) = \begin{cases} P(w_i \mid w_{i-1}), & \text{if } C(w_{i-1}, w_i) \neq 0, \\ \alpha P(w_i), & \text{otherwise.} \end{cases}$$

So far, this model does not tell us how to estimate the n-gram probabilities to the right of the formula. A first idea would be to use the maximum likelihood estimate for bigrams and unigrams. With $\alpha = 1$, this corresponds to:

$$P_{\text{Backoff}}(w_i \mid w_{i-1}) = \begin{cases} P_{\text{MLE}}(w_i \mid w_{i-1}) = \dfrac{C(w_{i-1}, w_i)}{C(w_{i-1})}, & \text{if } C(w_{i-1}, w_i) \neq 0, \\ P_{\text{MLE}}(w_i) = \dfrac{C(w_i)}{\#\text{words}}, & \text{otherwise.} \end{cases}$$

and Table 7.7 shows the probability estimates we can derive from our small corpus. They yield a sentence probability of 2.11×10^{-40} for our example.

This back-off technique is relatively easy to implement and Brants et al. (2007) applied it to 5-grams on a corpus of three trillion tokens with a back-off factor $\alpha = 0.4$. They used the recursive definition:

$$P_{\text{Backoff}}(w_i \mid w_{i-k}, ..., w_{i-1}) = \begin{cases} P_{\text{MLE}}(w_i \mid w_{i-k}, ..., w_{i-1}), & \text{if } C(w_{i-k}, ..., w_i) \neq 0, \\ \alpha P_{\text{Backoff}}(w_i \mid w_{i-k+1}, ..., w_{i-1}), & \text{otherwise.} \end{cases}$$

However, the result is not a probability as the sum of all the probabilities, $\sum_{w_i} P(w_i|w_{i-1})$, can be greater than 1. In the next section, we describe Katz's (1987) back-off model that provides an efficient and elegant solution to this problem.

7.5.3 Katz's Back-off Model

As with linear interpolation in Sect. 7.5.1, back-off combines n-grams of variable length while keeping a probability sum of 1. This means that for a bigram language model, we need to discount the bigram estimates to make room for the unigrams and then weight these unigrams to ensure that the sum of probabilities is equal to 1. This is precisely the definition of Katz's model, where Katz (1987) replaced the maximum likelihood estimates for bigrams with Good–Turing's estimates:

$$P_{\text{Katz}}(w_i|w_{i-1}) = \begin{cases} \tilde{P}(w_i|w_{i-1}), & \text{if } C(w_{i-1}, w_i) \neq 0, \\ \alpha P(w_i), & \text{otherwise.} \end{cases}$$

We first use the Good–Turing estimates to discount the observed bigrams,

$$\tilde{P}(w_i|w_{i-1}) = \frac{c^*(w_{i-1}, w_i)}{C(w_{i-1})},$$

for instance, with the values in Tables 7.4 and 7.5 for our sentence. We then assign the remaining probability mass to the unigrams.

To compute α, we add the two terms of Katz's back-off model, the discounted probabilities of the observed bigrams, and, for the unseen bigrams, the weighted unigram probabilities:

$$\begin{aligned} \sum_{w_i} P_{\text{Katz}}(w_i|w_{i-1}) &= \sum_{w_i, C(w_{i-1}, w_i) > 0} \tilde{P}(w_i|w_{i-1}) + \alpha \sum_{w_i, C(w_{i-1}, w_i) = 0} P_{\text{MLE}}(w_i), \\ &= 1. \end{aligned}$$

We know that this sum equals 1, and we derive α from it:

$$\alpha = \alpha(w_{i-1}) = \frac{1 - \sum_{w_i, C(w_{i-1}, w_i) > 0} \tilde{P}(w_i|w_{i-1})}{\sum_{w_i, C(w_{i-1}, w_i) = 0} P_{\text{MLE}}(w_i)}.$$

For trigrams or n-grams of higher order, we apply Katz's model recursively:

$$P_{\text{Katz}}(w_i|w_{i-2}, w_{i-1}) = \begin{cases} \tilde{P}(w_i|w_{i-2}, w_{i-1}), & \text{if } C(w_{i-2}, w_{i-1}, w_i) \neq 0, \\ \alpha(w_{i-2}, w_{i-1}) P_{\text{Katz}}(w_i|w_{i-1}), & \text{otherwise.} \end{cases}$$

7.6 Industrial *N*-grams

The Internet made it possible to put together collections of *n*-gram of a size unimaginable a few years ago. Examples of such collections include the Google *n*-grams (Franz and Brants, 2006) and Microsoft Web *n*-gram service (Huang et al., 2010; Wang et al., 2010).

The Google *n*-grams were extracted from a corpus of one trillion words and include unigram, bigram, trigram, 4-gram, and 5-gram counts. The excerpt below shows an example of trigram counts:

```
ceramics collectables collectibles 55
ceramics collectables fine 130
ceramics collected by 52
ceramics collectible pottery 50
ceramics collectibles cooking 45
ceramics collection , 144
ceramics collection . 247
ceramics collection </S> 120
ceramics collection and 43
```

Both companies, Google and Microsoft, use these *n*-grams in a number of applications and made them available to the public as well.

7.7 Quality of a Language Model

7.7.1 Intuitive Presentation

We can compute the probability of sequences of any length or of whole texts. As each word in the sequence corresponds to a conditional probability less than 1, the product will naturally decrease with the length of the sequence. To make sense, we normally average it by the number of words in the sequence and extract its *n*th root. This measure, which is a sort of a per-word probability of a sequence L, is easier to compute using a logarithm:

$$H(L) = -\frac{1}{n}\log_2 P(w_1,...,w_n).$$

We have seen that trigrams are better predictors than bigrams, which are better than unigrams. This means that the probability of a very long sequence computed with a bigram model will normally be higher than with a unigram one. The log measure will then be lower.

Intuitively, this means that the $H(L)$ measure will be a quality marker for a language model where lower numbers will correspond to better models. This intuition has mathematical foundations, as we will see in the two next sections.

7.7.2 Entropy Rate

We used entropy with characters in Chap. 4. We can use it with any symbols such as words, bigrams, trigrams, or any n-grams. When we normalize it by the length of the word sequence, we define the **entropy rate**:

$$H(L) = -\frac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1,...,w_n) \log_2 p(w_1,...,w_n),$$

where L is the set of all possible sequences of length n.

It has been proven that when $n \to \infty$ or n is very large and under certain conditions, we have

$$\begin{aligned} H(L) &= \lim_{n\to\infty} -\frac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1,...,w_n) \log_2 p(w_1,...,w_n), \\ &= \lim_{n\to\infty} -\frac{1}{n} \log_2 p(w_1,...,w_n), \end{aligned}$$

which means that we can compute $H(L)$ from a very long sequence, ideally infinite, instead of summing of all the sequences of a definite length.

7.7.3 Cross Entropy

We can also use cross entropy, which is measured between a text, called the language and governed by an unknown probability p, and a language model m. Using the same definitions as in Chap. 4, the cross entropy of m on p is given by:

$$H(p,m) = -\frac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1,...,w_n) \log_2 m(w_1,...,w_n).$$

As for the entropy rate, it has been proven that, under certain conditions

$$\begin{aligned} H(p,m) &= \lim_{n\to\infty} -\frac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1,...,w_n) \log_2 m(w_1,...,w_n), \\ &= \lim_{n\to\infty} -\frac{1}{n} \log_2 m(w_1,...,w_n). \end{aligned}$$

In applications, we generally compute the cross entropy on the complete word sequence of a test set, governed by p, using a bigram or trigram model, m, derived from a training set.

In Chap. 4, we saw the inequality $H(p) \leq H(p,m)$. This means that the cross entropy will always be an upper bound of $H(p)$. As the objective of a language model is to be as close as possible to p, the best model will be the one yielding the lowest possible value. This forms the mathematical background of the intuitive presentation in Sect. 7.7.1.

7.7.4 Perplexity

The perplexity of a language model is defined as:

$$PP(p,m) = 2^{H(p,m)}.$$

Perplexity is interpreted as the average *branching factor* of a word: the statistically weighted number of words that follow a given word. Perplexity is equivalent to entropy. The only advantage of perplexity is that it results in numbers more comprehensible for human beings. It is therefore more popular to measure the quality of language models. As is the case for entropy, the objective is to minimize it: the better the language model, the lower the perplexity.

7.8 Collocations

Collocations are recurrent combinations of words. Palmer (1933), one of the first to study them comprehensively, defined them as:

> succession[s] of two or more words that must be learnt as an integral whole and not pieced together from its component parts

or as *comings-together-of-words*. Collocations are ubiquitous and arbitrary in English, French, German, and other languages. Simplest collocations are fixed n-grams such as *The White House* and *Le Président de la République*. Other collocations involve some morphological or syntactic variation such as the one linking *make* and *decision* in American English: *to make a decision*, *decisions to be made*, *make an important decision*.

Collocations underlie word preferences that most of the time cannot easily be explained by a syntactic or semantic reasoning: they are merely resorting to usage. As a teacher of English in Japan, Palmer (1933) noted their importance for language learners. Collocations are in the mind of a native speaker. S/he can recognize them as valid. On the contrary, nonnative speakers may make mistakes when they are not aware of them or try to produce word-for-word translations. For this reason, many second language learners' dictionaries describe most frequent associations. In English, the *Oxford Advanced Learner's Dictionary*, *The Longman Dictionary of Contemporary English*, and *The Collins COBUILD* carefully list verbs and prepositions or particles commonly associated such as phrasal verbs *set up*, *set off*, and *set out*.

Lexicographers used to identify collocations by introspection and by observing corpora, at the risk of forgetting some of them. Statistical tests can automatically extract associated words or "sticky" pairs from raw corpora. We introduce three of these tests in this section together with programs in Python to compute them.

Table 7.8. Collocates of *surgery* extracted from the Bank of English using the mutual information test. Note the misspelled word *pioneeing*

Word	Frequency	Bigram word + *surgery*	Mutual information
arthroscopic	3	3	11.822
pioneeing	3	3	11.822
reconstructive	14	11	11.474
refractive	6	4	11.237
rhinoplasty	5	3	11.085

7.8.1 Word Preference Measurements

Mutual Information.

Mutual information (Fano, 1961) is a statistical measure that is widely used to quantify the strength of word associations (Church and Hanks, 1990).[1] Mutual information for the bigram w_i, w_j is defined as:

$$I(w_i, w_j) = \log_2 \frac{P(w_i, w_j)}{P(w_i)P(w_j)},$$

which will be positive, if the two words occur more frequently together than separately, equal to zero, if they are independent, in this case $P(w_i, w_j) = P(w_i)P(w_j)$, and negative, if they occur less frequently together than separately.

Using the maximum likelihood estimate, this corresponds to:

$$\begin{aligned} I(w_i, w_j) &= \log_2 \frac{N^2}{N-1} \cdot \frac{C(w_i, w_j)}{C(w_i)C(w_j)}, \\ &\approx \log_2 \frac{N \cdot C(w_i, w_j)}{C(w_i)C(w_j)}, \end{aligned}$$

where $C(w_i)$ and $C(w_j)$ are, respectively, the frequencies of word w_i and word w_j in the corpus, $C(w_i, w_j)$ is the frequency of bigram w_i, w_j, and N is the total number of words in the corpus.

Instead of just bigrams, where $j = i + 1$, we can count the number of times the two words w_i and w_j occur together sufficiently close, but not necessarily adjacently. $C(w_i, w_j)$ is then the number of times the word w_i is followed of preceded by w_j in a window of k words, where k typically ranges from 1 to 10, or within a sentence.

Table 7.8 shows collocates of the word *surgery*. High mutual information tends to show pairs of words occurring together but generally with a lower frequency, such as technical terms.

[1] Some authors now use the term *pointwise mutual information* to mean *mutual information*. Neither Fano (1961, pp. 27-28) nor Church and Hanks (1990) used this term and we kept the original one.

Table 7.9. Collocates of *set* extracted from Bank of English using the *t*-score

Word	Frequency	Bigram *set* + word	*t*-score
up	134,882	5512	67.980
a	1,228,514	7296	35.839
to	1,375,856	7688	33.592
off	52,036	888	23.780
out	12,3831	1252	23.320

t-Scores.

Given two words, the t-score (Church and Mercer, 1993) compares the hypothesis that the words form a collocation with the *null hypothesis* that posits that the cooccurrence is only governed by chance, that is $P(w_i, w_j) = P(w_i) \times P(w_j)$.

The t-score computes the difference between the two hypotheses, respectively, $mean(P(w_i, w_j))$ and $mean(P(w_i))mean(P(w_j))$, and divides it by the variances. It is defined by the formula:

$$t(w_i, w_j) = \frac{mean(P(w_i, w_j)) - mean(P(w_i))mean(P(w_j))}{\sqrt{\sigma^2(P(w_i, w_j)) + \sigma^2(P(w_i)P(w_j))}}.$$

The hypothesis that w_i and w_j are a collocation gives us a mean of $\frac{C(w_i, w_j)}{N}$; with the null hypothesis, the mean product is $\frac{C(w_i)}{N} \times \frac{C(w_j)}{N}$; and using a binomial assumption, the denominator is approximated to $\sqrt{\frac{C(w_i, w_j)}{N^2}}$. We have then:

$$t(w_i, w_j) = \frac{C(w_i, w_j) - \frac{1}{N}C(w_i)C(w_j)}{\sqrt{C(w_i, w_j)}}.$$

Table 7.9 shows collocates of *set* extracted from the Bank of English using the t-score. High t-scores show recurrent combinations of grammatical or very frequent words such as *of the*, *and the*, etc. Church and Mercer (1993) hint at the threshold value of 2 or more.

Likelihood Ratio.

Dunning (1993) criticized the t-score test and proposed an alternative measure based on binomial distributions and likelihood ratios. Assuming that the words have a binomial distribution, we can express the probability of having k counts of a word w in a sequence of N words knowing that w's probability is p as:

$$f(k; N, p) = \binom{N}{k} p^k (1-p)^{N-k},$$

where

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}.$$

The formula reflects the probability of having k counts of a word w, p^k, and $N-k$ counts of not having w, $(1-p)^{N-k}$. The binomial coefficient $\binom{N}{k}$ corresponds to the number of different ways of distributing k occurrences of the word w in a sequence of N words.

In the case of collocations, rather than measuring the distribution of single words, we want to evaluate the likelihood of the $w_i w_j$ bigram distribution. To do this, we can reformulate the binomial formula considering the word preceding w_j, which can either be w_i or a different word that we denote $\neg w_i$.

Let n_1 be the count of w_i and k_1, the count of the bigram $w_i w_j$ in the word sequence (the corpus). Let n_2 be the count of $\neg w_i$, and k_2, the count of the bigram $\neg w_i w_j$, where $\neg w_i w_j$ denotes a bigram in which the first word is not w_i and the second word is w_j. Let p_1 be the probability of w_j knowing that we have w_i preceding it, and p_2 be the probability of w_j knowing that we have $\neg w_i$ before it. The binomial distribution of observing the pairs $w_i w_j$ and $\neg w_i w_j$ in our sequence is:

$$f(k_1;n_1,p_1)f(k_2;n_2,p_2) = \binom{n_1}{k_1} p_1^{k_1}(1-p_1)^{n_1-k_1} \binom{n_2}{k_2} p_2^{k_2}(1-p_2)^{n_2-k_2}.$$

The basic idea to evaluate the collocation strength of a bigram $w_i w_j$ is to test two hypotheses:

- The two words w_i and w_j are part of a collocation. In this case, we will have $p_1 = P(w_j|w_i) \neq p_2 = P(w_j|\neg w_i)$ (Dependence hypothesis, H_{dep}).
- The two words w_i and w_j occur independently. In this case, we will have $p_1 = P(w_j|w_i) = p_2 = P(w_j|\neg w_i) = P(w_j) = p$ (Independence hypothesis, H_{ind}).

The logarithm of the hypothesis ratio corresponds to:

$$\begin{aligned}
-2\log\lambda &= 2\log\frac{H_{dep}}{H_{ind}}, \\
&= 2\log\frac{f(k_1;n_1,p_1)f(k_2;n_2,p_2)}{f(k_1;n_1,p)f(k_2;n_2,p)}, \\
&= 2(\log f(k_1;n_1,p_1) + \log f(k_2;n_2,p_2) - \log f(k_1;n_1,p) \\
&\quad - \log f(k_2;n_2,p)),
\end{aligned}$$

where $k_1 = C(w_i,w_j)$, $n_1 = C(w_i)$, $k_2 = C(w_j) - C(w_i,w_j)$, $n_2 = N - C(w_i)$, and $\log f(k;N,p) = k\log p + (N-k)\log(1-p)$.

Using the counts in Table 7.10 and the maximum likelihood estimate, we have

$$\begin{aligned}
p &= P(w_j) &&= \frac{C(w_j)}{N}, \\
p_1 &= P(w_j|w_i) &&= \frac{C(w_i,w_j)}{C(w_i)}, \text{and} \\
p_2 &= P(w_j|\neg w_i) &&= \frac{C(w_j) - C(w_i,w_j)}{N - C(w_i)},
\end{aligned}$$

Table 7.10. A contingency table containing bigram counts, where $\neg w_i w_j$ represents bigrams in which the first word is not w_i and the second word is w_j. N is the number of words in the corpus

	$\mathbf{w_i}$	$\neg\mathbf{w_i}$
$\mathbf{w_j}$	$C(w_i, w_j)$	$C(\neg w_i, w_j) = C(w_j) - C(w_i, w_j)$
$\neg\mathbf{w_j}$	$C(w_i, \neg w_j) = C(w_i) - C(w_i, w_j)$	$C(\neg w_i, \neg w_j) = N - C(w_i, w_j)$

where N is the number of words in the corpus.

7.8.2 Extracting Collocations with Python

The three measurements, mutual information, t-scores, and likelihood ratio, use unigram and bigram statistics. To compute them, we first tokenize the text, and count words and bigrams using the functions we have described in Sects. 6.4.3 and 7.2.2. We also need the number of words in the corpus. This corresponds to the size of the `words` list: `len(words)`.

Mutual Information.

The mutual infomation function iterates over the `freq_bigrams` list and applies the mutual information formula:

```
def mutual_info(words, freq_unigrams, freq_bigrams):
    mi = {}
    factor = len(words) * len(words) / (len(words) - 1)
    for bigram in freq_bigrams:
        mi[bigram] = (
            math.log(factor * freq_bigrams[bigram] /
                     (freq_unigrams[bigram[0]] *
                      freq_unigrams[bigram[1]]), 2))
    return mi
```

To run the computation, we first tokenize the text and collect the unigram and bigram frequencies. We then call `mutual_info()` and print the results:

```
if __name__ == '__main__':
    text = sys.stdin.read().lower()
    words = tokenize(text)
    frequency = count_unigrams(words)
    frequency_bigrams = count_bigrams(words)
    mi = mutual_info(words, frequency, frequency_bigrams)

    for bigram in sorted(mi.keys(), key=mi.get):
        print(mi[bigram], '\t', bigram, '\t',
```

```
            frequency[bigram[0]], '\t',
            frequency[bigram[1]], '\t',
            frequency_bigrams[bigram])
```

It is a common practice to apply a cutoff to the bigram frequencies. If we only want to consider bigrams that occur 10 times or more in the corpus, we just insert this statement in the beginning of the `for` loop:

```
if frequency_bigrams[bigram] < 10: continue
```

t-Scores.

The program is similar to the previous one except the formula:

```
def t_scores(words, freq_unigrams, freq_bigrams):
    ts = {}
    for bigram in freq_bigrams:
        ts[bigram] = ((freq_bigrams[bigram] -
                       freq_unigrams[bigram[0]] *
                       freq_unigrams[bigram[1]] /
                       len(words)) /
                       math.sqrt(freq_bigrams[bigram]))
    return ts
```

We use the same sequence of function calls as in `mutual_info()` to tokenize the text and collect the unigram and bigram frequencies.

Log Likelihood Ratio.

The program is similar to the previous one except the formula:

```
def likelihood_ratio(words, freq_unigrams, freq_bigrams):
    lr = {}
    for bigram in freq_bigrams:
        p = freq_unigrams[bigram[1]] / len(words)
        p1 = freq_bigrams[bigram] / freq_unigrams[bigram[0]]
        p2 = ((freq_unigrams[bigram[1]] - freq_bigrams[bigram])
             / (len(words) - freq_unigrams[bigram[0]]))
        if p1 != 1.0 and p2 != 0.0:
            lr[bigram] = 2.0 * (
                log_f(freq_bigrams[bigram],
                      freq_unigrams[bigram[0]], p1) +
                log_f(freq_unigrams[bigram[1]] -
                      freq_bigrams[bigram],
                      len(words) - freq_unigrams[bigram[0]], p2) -
                log_f(freq_bigrams[bigram],
                      freq_unigrams[bigram[0]], p) -
```

Rank	Mutual information	t-scores	Log-likelihood
1	rosy fingered	of the	son of
2	mixing bowl	son of	of the
3	barley meal	in the	the achaeans
4	fingered dawn	the achaeans	i am
5	dawn appeared	the trojans	the trojans
6	thigh bones	he was	he was
7	outer court	as he	at once
8	aegis bearing	to the	as he
9	morning rosy	on the	i will
10	ox hide	i will	you are

Table 7.11. The 10 strongest collocations according to three measures on a corpus consisting of *the Iliad* and *the Odyssey*. We applied a cutoff of 15 for mutual information

```
                log_f(freq_unigrams[bigram[1]] -
                      freq_bigrams[bigram],
                      len(words) - freq_unigrams[bigram[0]], p))
    return lr

def log_f(k, N, p):
    return k * log(p) + (N - k) * log(1 - p)
```

7.8.3 Applying Collocation Measures

To observe concretely the collocations extracted by the three measures, we applied them to a small corpus consisting of *the Iliad* and *the Odyssey*. These two texts are easy to obtain from the internet and we used translations from Samuel Butler from which we only kept the text, removing the title, translator name, and copyright statements. We set the text in lowercase, tokenized it, and ran the programs from Sect. 7.8.2. Table 7.11 shows the results with a comparison of the 10 strongest collocations according to mutual information, t-scores, and log-likelihood.

Mutual information reaches a maximum with pairs that are unique, where the first and second members of the pairs are also unique. These unique words are not really significant; we then applied a cutoff to discard te pairs that have less than 15 occurrences in the corpus. The collocations in Table 7.11 obtained by mutual information reflect concepts or terms from the world of Homer with pairs like *rosy fingered*, *barley meal*, or *ox hide*. This is not surprising given the input corpus.

Table 7.11 also shows that t-scores and log-likelihood have a significant overlap for this corpus, seven pairs out of 10, while mutual information yields a completely different list that shares no collocation with the two other measures. This is a general property that can be observed on other corpora.

7.9 Word Clustering

Mutual information can be used in many situations, where one needs to quantify the interdependence of two elements. Word clustering is one of them. Noticing that words like *Thursday* and *Friday* are interchangeable in many contexts such as

The meeting next Thursday,

Brown et al. (1992) designed an algorithm to augment n-gram language models with classes. In our example, the days of the week form an intuitive class that will make the estimation of the trigram probability

$$P(\text{meeting next Thursday})$$

less subject to the n-gram sparsity as it will be able to use the words: *Monday*, *Tuesday*, *Wednesday*, etc. and not only *Thursday*.

Given a corpus, Brown et al. (1992) proposed to build the classes from a partition of the vocabulary, where every word belongs to one of q classes and where q is a parameter of the clustering algorithm. Similarly to an n-gram language model, Brown et al. (1992) defined an n-gram class model by the equality:

$$P(w_k|w_1,...,w_{k-1}) = P(w_k|\ c_k)P(c_k|\ c_1,...,c_{k-1}),$$

where $c_1,...,c_k$ is the sequence of classes such that $w_i \in c_i$. For a bigram model, we can rewrite this equation as:

$$P(w_k|\ w_{k-1}) = P(w_k|c_k)P(c_k|c_{k-1}).$$

7.9.1 The Optimal Partition

Now that we have a new language model that takes classes into account, we may wonder how to find them practically. Brown et al. (1992) proposed to use the opposite of the entropy rate computed over the corpus L to assess the quality of a partition π. When using a bigram class model, this corresponds to:

$$\begin{aligned} -H(L,\pi) &= \frac{1}{n}\log_2 P(w_1,...,w_n), \\ &= \frac{1}{n}\log_2 \prod_{i=1}^{n} P(w_i|w_{i-1}), \\ &= \frac{1}{n}\log_2 \prod_{i=1}^{n} P(w_i|c_i)P(c_i|c_{i-1}), \end{aligned}$$

where we obtain the optimal partition by maximizing this quantity.

We can rewrite $H(L,\pi)$ and express it in the form of average mutual information between adjacent classes:

$$\begin{aligned}
\frac{1}{n}\log_2\prod_{i=1}^{n}P(w_i|c_i)P(c_i|c_{i-1}) &= \frac{1}{n}\log_2\prod_{i=1}^{n}\frac{C(w_i)}{C(c_i)}\cdot\frac{C(c_{i-1},c_i)}{C(c_{i-1})},\\
&= \frac{1}{n}\log_2\prod_{i=1}^{n}\frac{C(w_i)}{n}\cdot\frac{n\cdot C(c_{i-1},c_i)}{C(c_{i-1})\cdot C(c_i)},\\
&= \frac{1}{n}\log_2\prod_{i=1}^{n}P(w_i)\cdot\frac{P(c_{i-1},c_i)}{P(c_{i-1})\cdot P(c_i)},\\
&= \frac{1}{n}\sum_{i=1}^{n}\log_2 P(w_i)+\frac{1}{n}\sum_{i=1}^{n}\log_2\frac{P(c_{i-1},c_i)}{P(c_{i-1})\cdot P(c_i)}.
\end{aligned}$$

Eventually, we rewrite the entropy rate of a bigram class model as:

$$-H(L,\pi)=\sum_{c_i,c_j}P(c_i,c_j)\log_2\frac{P(c_i,c_j)}{P(c_i)P(c_j)}+\sum_{w}P(w)\log_2 P(w),$$

where w is a word of the corpus and c_i,c_j, a class bigram. The first term of $H(L,\pi)$ is the **average mutual information** (Fano, 1961, pp. 40-42) between adjacent classes and the second one, the opposite of the entropy of the word distribution. As this entropy is constant whatever the partition, we can ignore it. The optimal partition then corresponds to the maximal average mutual information between classes:

$$\sum_{c_i,c_j}P(c_i,c_j)\log_2\frac{P(c_i,c_j)}{P(c_i)P(c_j)}.$$

Fano (1961) showed that this average mutual information is always positive or equal to zero.

7.9.2 Building the Partition

Starting from a value of q, for instance 1,000 classes, a brute force solution to find the optimal partition would be to generate all the possible partitions of the vocabulary into 1,000 sets, compute the entropy rate for each partition, and keep the highest one. As you may have guessed, this is far from possible. The number of partitions of a vocabulary of 20,000 words into 1,000 classes is given by the Stirling number of the second kind, $S(20000,1000)$, an astronomic number that exceeds the capacity of any computer on earth.

Fortunately, there are possible approximations. Brown et al. (1992) proposed an iterative, greedy algorithm to build the partition, that, at each step of the clustering process, merges one pair of classes. The algorithm starts from an initial partition, where each word defines a distinct class and stops when it has reached the desired number of classes. For a corpus with a vocabulary of V distinct words to be clustered into q classes, we have V classes when we start, $V-1$ classes at step 1, $V-2$ at step 2, and finally q classes after $V-q$ steps.

At a given step k of the clustering process, the algorithm generates all the possible pairs of classes; for each pair, it computes the average mutual information that would result from merging the pair; and it merges the pair of classes that leads to the highest

average mutual information. The complexity of this algorithm is of order V^5, which makes it impractical for most corpora.

This first implementation unnecessarily repeats computations from step to step and it is possible to reduce its complexity. We denote I_k, the average mutual information at step k; $I_k(c_i,c_j)$, the average mutual information after we merge c_i and c_j; $L_k(c_i,c_j) = I_k(c_i,c_j) - I_k$, the difference (or loss), which we want to maximize; and, using a notation by Liang (2005), $w(c_i,c_j)$, the term brought by two adjacent classes c_i and c_j:

$$w(c_i,c_j) = \begin{cases} P(c_i,c_j)\log_2 \dfrac{P(c_i,c_j)}{P(c_i)P(c_j)} + P(c_j,c_i)\log_2 \dfrac{P(c_j,c_i)}{P(c_i)P(c_j)}, & \text{if } c_i \neq c_j, \\ P(c_i,c_j)\log_2 \dfrac{P(c_i,c_j)}{P(c_i)P(c_j)}, & \text{if } c_i = c_j, \end{cases}$$

where $i < j$. Denoting π_k, the partition at step k and $\pi_k(c_i,c_j)$, the partition after we merge c_i and c_j, where $\pi_k(c_i,c_j) = \pi_k - \{c_i,c_j\} + \{c_i \cup c_j\}$, we have:

$$L_k(c_i,c_j) = \sum_{d \in \pi_k(c_i,c_j)} w(c_i \cup c_j, d) - \sum_{d \in \pi_k} (w(c_i,d) + w(c_j,d)) + w(c_i,c_j).$$

After an initialization step, where π_0 consists of a partition of the corpus in word singletons, we determine π_1 that corresponds to:

$$\arg\max_{c_i,c_j} L_0(c_i,c_j),$$

then π_2, π_3, and so on. $L_k(c_i,c_j)$ can be effectively updated from step k to step $k+1$ for the classes that are not merged or recomputed for the pairs that are merged. The complexity is then reduced to V^3, which is still too high for medium to large corpora.

Brown et al. (1992) proposed a final optimization, where the words in the corpus are ordered by frequency and the q most frequent ones define q initial classes. The algorithm then processes each subsequent word from the rest of the list:

1. It creates a new class from the head of this list: a single word; we have $q+1$ classes at this point, and
2. it merges the pair of classes that maximizes the average mutual information; we have q classes again at the end of the step.

The algorithm needs $V - q$ steps to complete the clustering.

7.9.3 Examples of Word Clusters

Brown et al. (1992) applied their clustering algorithm to divide a vocabulary of more than 260,000 words into 1,000 classes. Table 7.12 shows examples of classes they obtained.

Table 7.12. Examples of classes obtained by the Brown clustering algorithm from a vocabulary of more than 260,000 words. After Brown et al. (1992)

Friday Monday Thursday Wednesday Tuesday Saturday Sunday weekends Sundays Saturdays
down backwards ashore sideways southward northward overboard aloft downwards adrift
great big vast sudden mere sheer gigantic lifelong scant colossal
John George James Bob Robert Paul William Jim David Mike

7.10 Dimensionality Reduction

The one-hot encoding technique we saw in Sect. 5.11 and the bags of words in Sect. 6.5.3 produce very large, sparse matrices, *i.e.* containing many zeros, when applied to any significant corpus. We can mitigate this sparsity using a **principal component analysis** that will reduce the dimensions of the observations and result in what is sometimes called **dense vectors**.

7.10.1 Data Sets

Although the dimensionality reduction is normally applied to word counts or cooccurrences, we will first use character counts. As corpus, we will consider the chapters of the *Salammbô* novel and their translations in English. This setting is both realistic and pretty compact. In Sect 5.4, we already used the counts of letter *A* to discriminate English and French. Table 7.13 shows the rest of the counts for all the characters, set to lower case, broken down by chapter; 30 in total.

In the next sections, we will apply the dimensionality reduction to these two data sets:

1. The counts in Table 5.1 that show the frequencies of letter *A* by chapter as well as all the letters, independently of their value (*A*, *B*, *C*, etc.). This small set has only two dimensions and is then easy to visualize; we will restrict it to the French chapters;
2. The counts of each character in Table 7.13 to have a complete example.

7.10.2 Singular Value Decomposition

In Table 7.13, there are as many as 40 characters: the 26 unaccented letters from *a* to *z* and the 14 French accented letters: à, â, æ, ç, è, é, ê, ë, î, ï, ô, œ, ù, and û. This means that we represent each chapter in our collection by a vector of 40 dimensions, where the chapter coordinates are the character counts. If we extend this representation to words, as it is done in most applications, this yields gigantic spaces reaching billions of dimensions.

Fortunately, it is possible to reduce these dimensions using a technique called **singular value decomposition** (SVD) that keeps the resulting vectors semantically

Ch.	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'	'p'	'q'	'r'	's'	't'	'u'	'v'	'w'	'x'	'y'	'z'	'à'	'â'	'æ'	'ç'	'è'	'é'
01_fr	2503	365	857	1151	4312	264	349	295	1945	65	4	1946	726	1896	1372	789	248	1948	2996	1938	1792	414	0	129	94	20	128	36	0	35	102	423
02_fr	2992	391	1006	1388	4993	319	360	350	2345	81	6	2128	823	2308	1560	977	281	2376	3454	2411	2069	499	0	175	89	23	136	50	1	28	147	513
03_fr	1042	152	326	489	1785	136	122	126	784	41	7	816	397	778	612	315	102	792	1174	856	707	147	0	42	31	7	39	9	0	10	49	194
04_fr	2487	303	864	1137	4158	314	331	287	2028	57	3	1796	722	1958	1318	773	274	2000	2792	2031	1734	422	0	138	81	27	110	43	0	22	138	424
05_fr	2014	268	645	949	3394	223	215	242	1617	67	3	1513	651	1547	1053	672	166	1601	2192	1736	1396	315	1	83	67	18	90	67	0	24	112	367
06_fr	2805	368	910	1266	4535	332	384	378	2219	97	3	1900	841	2179	1569	868	285	2205	3065	2293	1895	453	0	151	80	39	131	42	0	30	122	548
07_fr	5062	706	1770	2398	8512	623	622	620	4018	126	19	3726	1596	3851	2823	1532	468	4015	5634	4116	3518	844	0	272	148	71	246	50	1	46	232	966
08_fr	2643	325	869	1085	4229	307	317	359	2102	85	4	1857	811	2041	1367	833	239	2132	2814	2134	1788	437	0	135	64	30	130	43	0	34	119	502
09_fr	2126	289	771	920	3599	278	289	279	1805	52	6	1499	619	1711	1130	651	187	1719	2404	1763	1448	348	0	119	58	20	90	24	2	16	99	370
10_fr	1784	249	546	805	3002	179	202	215	1319	60	5	1462	598	1246	922	557	172	1242	1769	1423	1191	270	0	65	61	11	73	18	0	16	68	304
11_fr	2641	381	817	1078	4306	263	277	330	1985	114	0	1886	900	1966	1356	763	230	1912	2564	2218	1737	425	0	114	61	25	101	40	0	34	108	438
12_fr	2766	373	935	1237	4618	329	350	349	2273	65	2	1955	812	2285	1419	865	272	2276	3131	2274	1923	455	0	149	98	37	129	33	0	23	151	480
13_fr	5047	725	1730	2273	8678	648	566	642	3940	140	22	3746	1597	3984	2736	1550	425	4081	5599	4387	3480	767	0	288	119	41	209	55	3	61	237	940
14_fr	5312	689	1754	2149	8870	628	630	673	4278	143	2	3780	1610	4255	2713	1599	512	4271	5770	4467	3697	914	0	283	145	41	224	75	0	56	260	1019
15_fr	1215	173	402	582	2195	150	134	148	969	27	6	950	387	906	697	417	103	985	1395	1037	893	206	0	63	36	3	48	20	2	17	58	221
01_en	2217	451	729	1316	3967	596	662	2060	1823	22	200	1204	656	1851	1897	525	19	1764	1942	2547	704	258	653	29	401	18	0	0	0	0	0	0
02_en	2761	551	777	1548	4543	685	769	2530	2163	13	284	1319	829	2218	2237	606	21	2019	2411	3083	861	295	769	37	475	31	0	0	0	0	0	0
03_en	990	183	271	557	1570	279	253	875	783	4	82	520	333	816	828	194	13	711	864	1048	298	94	254	8	145	15	0	0	0	0	0	0
04_en	2274	454	736	1315	3814	595	559	1978	1835	22	198	1073	690	1771	1865	514	33	1726	1918	2704	745	245	663	60	467	19	0	0	0	0	0	0
05_en	1865	400	553	1135	3210	515	525	1693	1482	7	153	949	571	1468	1586	517	17	1357	1646	2178	663	194	568	26	330	33	0	0	0	0	0	0
06_en	2606	518	797	1509	4237	687	669	2254	2097	26	216	1239	763	2174	2231	613	25	1931	2192	2955	899	277	733	49	464	37	0	0	0	0	0	0
07_en	4805	913	1521	2681	7834	1366	1163	4379	3838	42	416	2434	1461	3816	4091	1040	39	3674	4060	5369	1552	465	1332	74	843	52	0	0	0	0	0	0
08_en	2396	431	702	1416	4014	621	624	2171	2011	24	216	1152	748	2085	1947	527	33	1915	1966	2765	789	266	695	65	379	24	0	0	0	0	0	0
09_en	1993	408	653	1096	3373	575	517	1766	1648	16	146	861	629	1728	1698	442	20	1561	1626	2442	683	208	560	25	328	18	0	0	0	0	0	0
10_en	1627	359	451	933	2690	477	409	1475	1196	7	131	789	506	1266	1369	325	23	1211	1344	1759	502	181	410	31	255	20	0	0	0	0	0	0
11_en	2375	437	643	1364	3790	610	644	2217	1830	16	217	1122	799	1833	1948	486	23	1720	1945	2424	767	246	632	20	457	39	0	0	0	0	0	0
12_en	2560	489	757	1566	4331	677	650	2348	2033	28	234	1102	746	2125	2105	581	32	1939	2152	3046	750	278	721	35	418	40	0	0	0	0	0	0
13_en	4597	987	1462	2689	7963	1254	1201	4278	3634	39	432	2281	1493	3774	3911	1099	49	3577	3894	5540	1379	437	1374	77	673	49	0	0	0	0	0	0
14_en	4871	948	1439	2799	8179	1335	1140	4534	3829	36	427	2218	1534	4053	3989	1019	36	3689	3946	5858	1490	539	1377	90	856	49	0	0	0	0	0	0
15_en	1119	229	335	683	1994	323	281	1108	912	9	112	579	351	924	1004	305	9	863	997	1330	310	108	330	14	150	9	0	0	0	0	0	0

Table 7.13. Character counts per chapter, where the fr and en suffixes designate the language, either French or English

	French															English														
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
a	2503	2992	1042	2487	2014	2805	5062	2643	2126	1784	2641	2766	5047	5312	1215	2217	2761	990	2274	1865	2606	4805	2396	1993	1627	2375	2560	4597	4871	1119
b	365	391	152	303	268	368	706	325	289	249	381	373	725	689	173	451	551	183	454	400	518	913	431	408	359	437	489	987	948	229
c	857	1006	326	864	645	910	1770	869	771	546	817	935	1730	1754	402	729	777	271	736	553	797	1521	702	653	451	643	757	1462	1439	33
d	1151	1388	489	1137	949	1266	2398	1085	920	805	1078	1237	2273	2149	582	1316	1548	557	1315	1135	1509	2681	1416	1096	933	1364	1566	2689	2799	68
e	4312	4993	1785	4158	3394	4535	8512	4229	3599	3002	4306	4618	8678	8870	2195	3967	4543	1570	3814	3210	4237	7834	4014	3373	2690	3790	4331	7963	8179	199
f	264	319	136	314	223	332	623	307	278	179	263	329	648	628	150	596	685	279	595	515	687	1366	621	575	477	610	677	1254	1335	32
g	349	360	122	331	215	384	622	317	289	202	277	350	566	630	134	662	769	253	559	525	669	1163	624	517	409	644	650	1201	1140	28
h	295	350	126	287	242	378	620	359	279	215	330	349	642	673	148	2060	2530	875	1978	1693	2254	4379	2171	1766	1475	2217	2348	4278	4534	110
i	1945	2345	784	2028	1617	2219	4018	2102	1805	1319	1985	2273	3940	4278	969	1823	2163	783	1835	1482	2097	3838	2011	1648	1196	1830	2033	3634	3829	91
j	65	81	41	57	67	97	126	85	52	60	114	65	140	143	27	22	13	4	22	7	26	42	24	16	7	16	28	39	36	
k	4	6	7	3	3	3	19	4	6	5	0	2	22	2	6	200	284	82	198	153	216	416	216	146	131	217	234	432	427	11
l	1946	2128	816	1796	1513	1900	3726	1857	1499	1462	1886	1955	3746	3780	950	1204	1319	520	1073	949	1239	2434	1152	861	789	1122	1102	2281	2218	57
m	726	823	397	722	651	841	1596	811	619	598	900	812	1597	1610	387	656	829	333	690	571	763	1461	748	629	506	799	746	1493	1534	35
n	1896	2308	778	1958	1547	2179	3851	2041	1711	1246	1966	2285	3984	4255	906	1851	2218	816	1771	1468	2174	3816	2085	1728	1266	1833	2125	3774	4053	92
o	1372	1560	612	1318	1053	1569	2823	1367	1130	922	1356	1419	2736	2713	697	1897	2237	828	1865	1586	2231	4091	1947	1698	1369	1948	2105	3911	3989	100
p	789	977	315	773	672	868	1532	833	651	557	763	865	1550	1599	417	525	606	194	514	517	613	1040	527	442	325	486	581	1099	1019	30
q	248	281	102	274	166	285	468	239	187	172	230	272	425	512	103	19	21	13	33	17	25	39	33	20	23	23	32	49	36	
r	1948	2376	792	2000	1601	2205	4015	2132	1719	1242	1912	2276	4081	4271	985	1764	2019	711	1726	1357	1931	3674	1915	1561	1211	1720	1939	3577	3689	86
s	2996	3454	1174	2792	2192	3065	5634	2814	2404	1769	2564	3131	5599	5770	1395	1942	2411	864	1918	1646	2192	4060	1966	1626	1344	1945	2152	3894	3946	99
t	1938	2411	856	2031	1736	2293	4116	2134	1763	1423	2218	2274	4387	4467	1037	2547	3083	1048	2704	2178	2955	5369	2765	2442	1759	2424	3046	5540	5858	133
u	1792	2069	707	1734	1396	1895	3518	1788	1448	1191	1737	1923	3480	3697	893	704	861	298	745	663	899	1552	789	683	502	767	750	1379	1490	31
v	414	499	147	422	315	453	844	437	348	270	425	455	767	914	206	258	295	94	245	194	277	465	266	208	181	246	278	437	539	10
w	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	653	769	254	663	568	733	1332	695	560	410	632	721	1374	1377	33
x	129	175	42	138	83	151	272	135	119	65	114	149	288	283	63	29	37	8	60	26	49	74	65	25	31	20	35	77	90	1
y	94	89	31	81	67	80	148	64	58	61	61	98	119	145	36	401	475	145	467	330	464	843	379	328	255	457	418	673	856	15
z	20	23	7	27	18	39	71	30	20	11	25	37	41	41	3	18	31	15	19	33	37	52	24	18	20	39	40	49	49	
à	128	136	39	110	90	131	246	130	90	73	101	129	209	224	48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
â	36	50	9	43	67	42	50	43	24	18	40	33	55	75	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
æ	0	1	0	0	0	0	1	0	2	0	0	0	3	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
ç	35	28	10	22	24	30	46	34	16	16	34	23	61	56	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
è	102	147	49	138	112	122	232	119	99	68	108	151	237	260	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
é	423	513	194	424	367	548	966	502	370	304	438	480	940	1019	221	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
ê	43	68	24	36	44	57	96	54	43	53	68	60	126	94	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
ë	1	0	0	0	1	0	2	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
î	17	20	12	15	11	15	42	11	8	15	26	13	32	28	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
ï	2	0	0	2	8	12	9	1	2	5	15	3	5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
ô	20	20	27	15	23	15	41	14	13	38	50	15	37	45	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
ù	14	9	4	6	18	14	30	6	5	3	7	11	24	21	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
û	7	9	7	4	15	15	38	8	15	10	9	14	30	21	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
œ	5	5	2	8	7	9	9	5	3	5	7	0	13	12	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 7.14. Character counts per chapter in French, left part, and English, right part

close to their original values. Let us apply this decomposition to the data in Table 7.13 and denote $\mathbf{X}$ the $m \times n$ matrix of the letter counts per chapter, in our case, $m = 30$ and $n = 40$. From the works of Beltrami (1873), Jordan (1874), and Eckart and Young (1936), we know we can rewrite $\mathbf{X}$ as:

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\intercal,$$

where $\mathbf{U}$ is a matrix of dimensions $m \times m$, Σ, a diagonal matrix of dimensions $m \times n$, and $\mathbf{V}$, a matrix of dimensions $n \times n$. The diagonal terms of Σ are called the **singular values** and are traditionally arranged by decreasing value.

The singular value decomposition finds a sequence of orthonormal vectors, where the rows, here *Salammbô*'s chapters, have a maximal variance. The columns of $\mathbf{V}$ are called the principal axes and the columns of $\mathbf{U}\Sigma$, the principal components. We have the properties: $\mathbf{U}\mathbf{U}^\intercal = \mathbf{I}$ and $\mathbf{V}\mathbf{V}^\intercal = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

7.10.3 Data Representation and Preprocessing

The numpy or sklearn Python libraries have SVD functions that we can call without bothering about the mathematical details. To store $\mathbf{X}$, we use a numpy array as in Sect. 5.11 and prior to the decomposition, we need to standardize the counts for each character by subtracting the mean from the counts and dividing them by the standard deviation. As with neural networks in Sect. 5.14.2, we apply a normalization before the standardization:

```
from sklearn.preprocessing import StandardScaler, Normalizer

X_norm = Normalizer().fit_transform(X)
X_scaled = StandardScaler().fit_transform(X_norm)
```

7.10.4 Singular Value Decomposition

To carry out the singular value decomposition, we can either use `numpy` or `scikit-learn`. In the code below, we use numpy and the `linalg.svd` function. The statement and the returned values follow the mathematical formulation. If `X` is the input matrix corresponding to the scaled and possibly normalized data, we have:

```
import numpy as np

U, s, Vt = np.linalg.svd(X, full_matrices=False)
Us = U @ np.diag(s)
```

where `s` contains the singular values of Σ. We do not need to compute the full squared $\mathbf{U}$ matrix as the values of Σ outside the diagonal are zero and we set `full_matrices=False`. We compute the new coordinates of the chapters by multiplying `U` and the diagonal matrix of the singular values: `Us`.

Counts of letter A

Before we go to the data in Table 7.13, let us restrict the case to letter A, the total count of characters, and the French chapters. To write the code, we just need to store the data in Table 5.1, left part, in `X`, standardize it, and apply the SVD. Here, we set aside the normalization. The decomposition returns:

$$\mathbf{U} = \begin{bmatrix} -0.0630 & 0.1783 \\ 0.0325 & -0.0225 \\ -0.3577 & 0.2233 \\ -0.0696 & -0.0228 \\ -0.1609 & 0.2348 \\ -0.0073 & -0.1429 \\ 0.4606 & 0.5228 \\ -0.0436 & -0.3491 \\ -0.1430 & -0.0429 \\ -0.2150 & -0.2087 \\ -0.0453 & -0.4232 \\ -0.0126 & 0.0115 \\ 0.4511 & 0.1455 \\ 0.4960 & -0.3645 \\ -0.3221 & 0.2605 \end{bmatrix}; \Sigma = \begin{bmatrix} 5.4764 & 0 \\ 0 & 0.0933 \end{bmatrix}; \mathbf{V} = \begin{bmatrix} 0.7071 & 0.7071 \\ 0.7071 & -0.7071 \end{bmatrix}.$$

The first column vector in $\mathbf{V}$ corresponds to the direction of the regression line given in Sect. 5.5, while the second one is orthogonal to it; Σ contains the singular values; and $\mathbf{U}\Sigma$ contains the coordinates of the chapters in the new system defined by $\mathbf{V}$. We project the points by simply setting some values of Σ to 0, here the second one:

$$\Sigma = \begin{bmatrix} 5.4764 & 0 \\ 0 & 0 \end{bmatrix}.$$

Figure 7.1 shows the results of the analysis on three panes. The left pane shows the original standardized data set; the middle pane, the rotated data set, corresponding to $\mathbf{U}\Sigma$, and the right pane, the projection on the first right singular vector. This corresponds to a reduction from 2 to 1 and is equivalent to a projection on the regression line.

Counts of all the characters

Now that we have seen how two dimensional data are rotated and projected, let us apply the decomposition to all the characters in Table 7.13. This decomposition results in a vector of 30 singular values ranging from 30.17 to 1.34 10^{-14} that we can relate to the amount of information brought by the corresponding direction.

Projecting the data on the plane associated to the two highest values reduces the dimensions from 30 to 2, while keeping a good deal of the variance between the

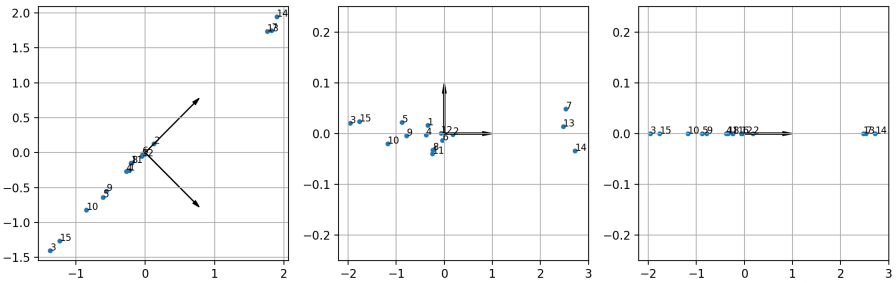

Fig. 7.1. Left pane: The standardized data set with the singular vectors; Middle pane: The rotated data set, note the change of scale; Right pane: The projection on the first right singular vector

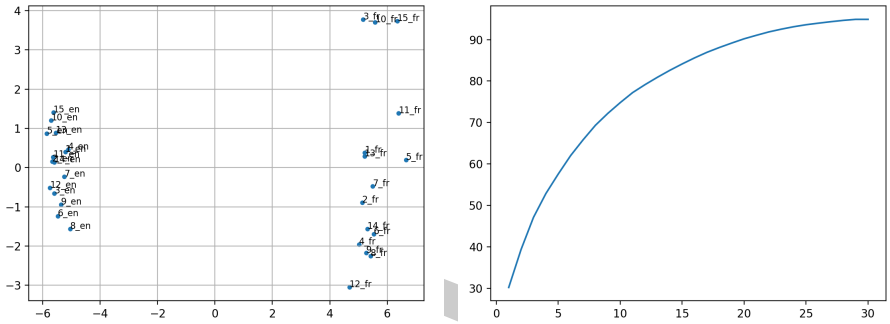

Fig. 7.2. Left pane: The chapters projected on a plane. The chapters in English are to the left on the x axis, while the French ones are to the right; Right pane: The cumulative sum of the singular values. The two first values are about 40% of the total

points. To carry this out, we just set all the singular values in Σ to 0 except the two first ones. Figure 7.2 shows the projected chapters corresponding to the rows in the truncated $\mathbf{U}\Sigma$ matrix.

The result is striking: On the left pane of the figure, the chapters form two relatively compact clusters, to the left and to the right of the x axis, corresponding to the English and French versions. On the right pane of the figure, we have the cumulative sum of the singular values: 39.37 for the two first ones and 94.90 for all.

The characters in a space of documents

In Table 7.13, we represented the chapters in a space of characters. We can do the opposite: Describe the characters in a space of documents. This is easy; we just need to transpose $\mathbf{X}$, apply the decomposition, and truncate $\mathbf{U}\Sigma$.

Figure 7.3 shows the results. Again, it enables us to visualize very quickly the properties of the characters relatively to French and English. Accented letters frequent in French like É or È, but rare in English are clustered in the bottom left

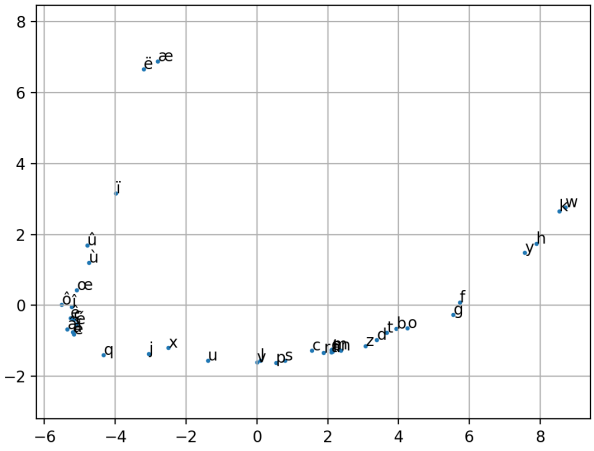

Fig. 7.3. Characters projected on a plane

corner; letters rare in French and English like Ë or Æ are in the left corner; letter that are rare in French, but frequent in English are on the right; while letter that are as frequent in French as in English are in the middle. So we can roughly interpret the x axis as a French to English direction; while the y axis is a sort of frequency axis.

As an alternative to `numpy.linalg.svd`, we could use `sklearn.decomposition.PCA`.

7.10.5 Word embeddings

We can extend the technique we described for characters to words. The matrix structure will be the same, but the rows will represent the words in the corpus, and the columns, documents, or more simply a context of a few words to the left and to the right of the focus word: w_i. A context C_j is then defined by a window of $2K$ words centered on the word:

$$w_{i-K}, w_{i-K+1}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+K-1}, w_{i+K},$$

where the context representation uses a bag of words. We can even reduce the context to a single word to the left or to the right of w_i and use bigrams.

We store the word-context pairs (w_i, C_j) in a matrix that is just the transpose of Table 6.8, where we replace the documents with a more general context, C. Each matrix element measures the association strength between word w_i and context C_j. In Table 7.13, we saw very simple pairs consisting of raw frequencies. In this section, we will use mutual information instead as we could see in Sect. 7.8 it produces relevant semantic associations.

Computing a complete matrix with the original definition of mutual information is not possible given that some pairs (w, C) have a count of 0, leading to an infinitely

Table 7.15. The word by context matrix. The context is either a document or a window of n words. Each cell (w_i, C_j) contains the mutual information of w_i and C_j. As unseen pairs have an undefined mutual information, we set their values to zero and we replace all the negative mutual informations with zeros too

D#\Words	C_1	C_2	C_3	...	C_n
w_1	$MI(w_1,C_1)$	$MI(w_1,C_2)$	$MI(w_1,C_3)$	...	$MI(w_1,C_n)$
w_2	$MI(w_2,C_1)$	$MI(w_2,C_2)$	$MI(w_2,C_3)$	...	$MI(w_2,C_n)$
w_3	$MI(w_3,C_1)$	$MI(w_3,C_2)$	$MI(w_3,C_3)$	...	$MI(w_3,C_n)$
...	...	...	...	...	...
w_m	$MI(w_m,C_1)$	$MI(w_m,C_2)$	$MI(w_m,C_3)$	...	$MI(w_m,C_n)$

negative logarithm. To solve this problem, Bullinaria and Levy (2007) proposed to set the mutual information to zero when the pairs are unseen and, to be consistent with the other pairs, to set pairs with a negative mutual information to zero too. This means that we keep the positive values, corresponding to associations that are more frequent than chance, and we set the rest to zeros. This work-around is very simple and produces a representation, where semantically related vectors are close in the vector space.

We compute the word embeddings with a singular value decomposition, where we truncate the matrix $\mathbf{U}\Sigma$ to 50, 100, 300, or 500 dimensions. The word embeddings are the rows of this matrix. We usually measure the similarity between two embeddings $\mathbf{u}$ and $\mathbf{v}$ with the cosine similarity:

$$\cos(\mathbf{u},\mathbf{v}) = \frac{\mathbf{u}\cdot\mathbf{v}}{||\mathbf{u}||\cdot||\mathbf{v}||},$$

ranging from -1 (most dissimilar) to 1 (most similar) or with the cosine distance ranging from 0 (closest) to 2 (most distant):

$$1-\cos(\mathbf{u},\mathbf{v}) = 1-\frac{\mathbf{u}\cdot\mathbf{v}}{||\mathbf{u}||\cdot||\mathbf{v}||}.$$

Instead of mutual information, we can use other similarity metrics to compute the matrix coefficients x_{ij} and carry out a dimensionality reduction:

- Benzécri (1981a) and Benzécri and Morfin (1981) defined a correspondence analysis method that applies the χ^2 metric to pairs of words or word-document pairs and a specific normalization of the matrices;
- Deerwester et al. (1990) used a formula similar to $tf \times idf$, defined in Sect. 6.5.3, consisting of a local weight, computed from a document, like the term frequency tf, and a global one, computed from the collection, like the inverted document frequency idf. They called this method **latent semantic indexing**.
  Dumais (1991) describes variants of the weights, where the best scheme to compute the coefficient x_{ij} corresponding to term i in document j is given by a local weight of $\log(tf_{ij}+1)$, where tf_{ij} is the frequency of term i in document j, and a global weight of

$$1 - \sum_j \frac{p_{ij} \log(p_{ij})}{\log(n\text{docs})},$$

where $p_{ij} = \dfrac{tf_{ij}}{gf_i}$, gf_i is the global frequency of term i: The total number of times it occurs in the collection, and $n$docs, the total number of documents.

7.11 Further Reading

Language models and statistical techniques were applied first to speech recognition, lexicography, and later to most domains of natural language processing. For a historical turning point in their popularity, see the special issues of *Computational Linguistics* (1993, 1 and 2). See also Norvig (2009) for beautiful Python programs building on language models to segment words, decipher codes, or check word spelling.

Interested readers will find additional details on language modeling techniques in Chen and Goodman (1998), and on χ^2 tests and likelihood ratios to improve collocation detection in Dunning (1993). Manning and Schütze (1999, Chapter 5) is a good reference on collocations, while Brown et al. (1992) describe other methods to create semantic clusters. Machine-learning techniques frequently use word features. Brown clustering can improve the classification performance when words are complemented with cluster features. Liang (2005) implemented an efficient version of this clustering algorithm that is available for download (`https://github.com/percyliang/brown-cluster`).

The cooccurrences matrices we saw in Sect. 7.10 express contextual or semantic similarities. Visualization and clustering inspired the first experiments carried out with singular value decomposition as the dimensions of these matrices make their comprehension beyond human capacity. For a description of early analyses, see Benzécri (1973a), Benzécri (1973b), and especially Benzécri (1981b), that contains studies covering multiple languages and genres with plenty of figures. This was the idea of word embeddings: To map discrete representations of words to continuous vectors that could be shown in a plane or a three-dimensional space. As the power of computers and graphical processing units (GPU) increased dramatically, word embeddings found new applications. Most notably, replacing (or complementing) input words with their embeddings is an efficient way to mitigate sparse data. This explains why they are now ubiquitous in classification models.

In this chapter, we saw word embeddings based on mutual information. There are other popular dimensionality reduction methods: They include word2vec (Mikolov et al., 2013), Glove (Pennington et al., 2014), and fastText (Bojanowski et al., 2017). These three systems use iterative training procedures and can handle larger corpora than singular value decomposition. Nonetheless, Levy et al. (2015) showed that all these methods produce roughly equivalent results. The main differences come from parameter settings.

The code of word2vec, Glove, and fastText is open source. In addition, the authors applied their programs to very large corpora and multiple languages, and pub-

lished the vectors. This means that we do not need to train these word embeddings, we can simply download them.

Exercises

7.1. Retrieve a text you like on the Internet. Give the five most frequent bigrams and trigrams.

7.2. Retrieve a text you like on the Internet. Divide it into a training set and a test set. Implement the Laplace rule either in Python. Learn the probabilities on the training set and compute the perplexity of the test set.

7.3. Retrieve a text you like on the Internet. Divide it into a training set and a test set. Implement the Good–Turing estimation either in Python. Learn the probabilities on the training set and compute the perplexity of the test set.

7.4. Implement the mutual information score with a word to the left and to the right of the focus word: unordered pairs.

7.5. Implement the mutual information score with a window of five words to the left and to the right of the word.

7.6. Select a corpus and compute the positive mutual information matrix for unordered pairs of words. Apply a singular value decomposition to this matrix and truncate the vectors to 50 dimensions.

7.7. Using the results from Exercise 7.6, select 10 words from the corpus and determine their 10 nearest neighbors using the cosine similarity.

Index

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA. USENIX Association.

Agnäs, M.-S., Alshawi, H., Bretan, I., Carter, D., Ceder, K., Collins, M., Crouch, R., Digalakis, V., Ekholm, B., Gambäck, B., Kaja, J., Karlgren, J., Lyberg, B., Price, P., Pulman, S., Rayner, M., Samuelsson, C., and Svensson, T. (1994). Spoken language translator, first-year report. Research Report R94:03, SICS, Kista, Sweden.

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In Holub, J. and Žd'árek, J., editors, *Implementation and Application of Automata. 12th International Conference on Implementation and Application of Automata, CIAA 2007, Prague, July 2007. Revised Selected Papers*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23, Berlin Heidelberg New York. Springer Verlag.

Allen, J. F. (1994). *Natural Language Understanding*. Benjamin/Cummings, Redwood City, California, second edition.

Allen, J. F., Schubert, L. K., Ferguson, G., Heeman, P., Hwang, C. H., Kato, T., Light, M., Martin, N. G., Miller, B. W., Poesio, M., and Traum, D. R. (1995). The TRAINS project: A case study in building a conversational planning agent. *Journal of Experimental and Theoretical AI*, 7:7–48.

Apache OpenNLP Development Community (2012). *Apache OpenNLP Developer Documentation*. The Apache Software Foundation, 1.5.2 edition.

Baeza-Yates, R. and Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley, New York, 2nd edition.

Ball, G., Ling, D., Kurlander, D., Miller, J., Pugh, D., Skelly, T., Stankosky, A., Thiel, D., Dantzich, M. V., and Wax, T. (1997). Lifelike computer characters: the Persona

project at Microsoft Research. In Bradshaw, J. M., editor, *Software Agents*, pages 191–222. AAAI Press/MIT Press, Cambridge, Massachusetts.

Beltrami, E. (1873). Sulle funzioni bilineari. *Giornale di matematiche*, 11:98–106.

Bentley, J., Knuth, D., and McIlroy, D. (1986). Programming pearls. *Communications of the ACM*, 6(29):471–483.

Benzécri, F. and Morfin, M. (1981). *Introduction à l'analyse des correspondances et à la classification automatique*, volume 3, pages 73–135. Dunod, Paris.

Benzécri, J.-P. (1973a). *L'Analyse des données: la taxinomie*, volume I. Dunod, Paris.

Benzécri, J.-P. (1973b). *L'Analyse des données: l'analyse des correspondances*, volume II. Dunod, Paris.

Benzécri, J.-P. (1981a). *Analyse statistique des données linguistiques*, volume 3, pages 3–45. Dunod, Paris.

Benzécri, J.-P., editor (1981b). *Pratique de l'analyse des données: Linguistique et lexicologie*, volume 3. Dunod, Paris.

Berkson, J. (1944). Application of the logistic function to bio-assay. *Journal of the American Statistical Association*, 39(227):357–365.

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Sebastopol, California.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Boscovich, R. J. (1770). *Voyage astronomique et géographique dans l'État de l'Église*. N. M. Tilliard, Paris.

Boser, B., Guyon, I., and Vapnik, V. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh. ACM.

Bourne, S. R. (1982). *The Unix System*. International computer science series. Addison-Wesley, London.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, page 858–867, Prague.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1–7):107–117. Proceedings of WWW7.

Brown, P. F., Della Pietra, V. J., deSouza, P. V., Lai, J. C., and Mercer, R. L. (1992). Class-based *n*-gram models of natural language. *Computational Linguistics*, 18(4):467–489.

Bullinaria, J. A. and Levy, J. P. (2007). Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods*, 3(39):510–526.

Busa, R. (1974). *Index Thomisticus: Sancti Thomae Aquinatis operum omnium indices et concordantiae in quibus verborum omnium et singulorum formae et lemmata cum suis frequentiis et contextibus variis modis referuntur*. Frommann-Holzboog, Stuttgart-Bad Cannstatt.

Busa, R. (1996). *Thomae Aquinatis Opera omnia cum hypertextibus in CD-ROM*. Editoria Elettronica Editel, Milan.

Busa, R. (2009). From punched cards to treebanks: 60 years of computational linguistics. In *Website of The Eighth International Workshop on Treebanks and Linguistic Theories*, Milan.

Cauchy, A. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, 25:536–538.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27.

Chen, S. F. and Goodman, J. (1998). An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Harvard University, Cambridge, Massachusetts.

Chollet, F. (2018). *Deep learning with Python*. Manning Publications, Shelter Island, New York.

Chomsky, N. (1957). *Syntactic structures*. Mouton, The Hague.

Church, K. W. and Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29.

Church, K. W. and Mercer, R. L. (1993). Introduction to the special issue on computational linguistics using large corpora. *Computational Linguistics*, 19(1):1–24.

Clarkson, P. R. and Rosenfeld, R. (1997). Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings ESCA Eurospeech*, Rhodes.

Cooper, D. (1999). Corpora: kwic concordances with Perl. CORPORA Mailing List Archive, Concordancing thread.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Covington, M. A. (1994). *Natural Language Processing for Prolog Programmers*. Prentice Hall, Upper Saddle River.

Crystal, D. (1997). *The Cambridge Encyclopedia of Language*. Cambridge University Press, Cambridge, 2nd edition.

d'Arc, S. J., editor (1970). *Concordance de la Bible, Nouveau Testament*. Éditions du Cerf – Desclées De Brouwer, Paris.

Davis, M. and Whistler, K. (2009). Unicode collation algorithm. Unicode Technical Standard 10, The Unicode Consortium. Version 5.2.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

Ducrot, O. and Schaeffer, J.-M., editors (1995). *Nouveau dictionnaire encyclopédique des sciences du langage*. Éditions du Seuil, Paris.

Dumais, S. T. (1991). Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236.

Dunning, T. (1993). Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74.

Eckart, C. and Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218.

Estoup, J.-B. (1912). *Gammes sténographiques: recueil de textes choisis pour l'acquisition méthodique de la vitesse*. Institut sténographique, Paris, 3e edition.

Fan, J., Kalyanpur, A., Gondek, D. C., and Ferrucci, D. A. (2012). Automatic knowledge extraction from documents. *IBM Journal of Research and Development*, 56(3.4):5:1 –5:10.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Fano, R. M. (1961). *Transmission of Information: A Statistical Theory of Communications*. MIT Press, New York.

Ferrucci, D. A. (2012). Introduction to "This is Watson". *IBM Journal of Research and Development*, 56(3.4):1:1 –1:15.

Francis, W. N. and Kucera, H. (1982). *Frequency Analysis of English Usage*. Houghton Mifflin, Boston.

Franz, A. and Brants, T. (2006). All our n-gram are belong to you. `http://googleresearch.blogspot.se/2006/08/all-our-n-gram-are-belong-to-you.html`. Retrieved 7 November 2013.

Fredholm, I. (1903). Sur une classe d'équations fonctionnelles. *Acta Mathematica*, 27:365–390.

Friedl, J. E. F. (2006). *Mastering Regular Expressions*. O'Reilly, Sebastopol, California, third edition.

Fromkin, V., editor (2000). *Linguistics: An Introduction to Linguistic Theory*. Blackwell, Oxford.

Fromkin, V., Rodman, R., and Hyams, N. (2010). *An Introduction to Language*. Wadsworth, Cengage Learning, Boston, 9th edition.

Gal, A., Lapalme, G., and Saint-Dizier, P. (1989). *Prolog pour l'analyse automatique du langage naturel*. Eyrolles, Paris.

Gal, A., Lapalme, G., Saint-Dizier, P., and Somers, H. (1991). *Prolog for Natural Language Processing*. Wiley, Chichester.

Galton, F. (1886). Regression towards mediocrity in hereditary stature. *Journal of the Anthropological Institute*, 15:246–263.

Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in Prolog: An Introduction to Computational Linguistics*. Addison-Wesley, Wokingham.

Godéreaux, C., Diebel, K., El Guedj, P.-O., Revolta, F., and Nugues, P. (1996). An interactive spoken dialog interface to virtual worlds. In Connolly, J. H. and Pemberton, L., editors, *Linguistic Concepts and Methods in CSCW*, Computer supported cooperative work, chapter 13, pages 177–200. Springer, Berlin Heidelberg New York.

Godéreaux, C., El Guedj, P.-O., Revolta, F., and Nugues, P. (1998). Ulysse: An interactive, spoken dialogue interface to navigate in virtual worlds. Lexical, syntactic, and semantic issues. In Vince, J. and Earnshaw, R., editors, *Virtual Worlds on the Internet*, chapter 4, pages 53–70, 308–312. IEEE Computer Society Press, Los Alamitos, California.

Good, I. J. (1953). The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(16):237–264.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. The MIT Press, Cambridge, Massachusetts.

Goyvaerts, J. and Levithan, S. (2012). *Regular Expressions Cookbook*. O'Reilly Media, Sebastopol, California, 2nd edition.

Grefenstette, G. and Tapanainen, P. (1994). What is a word, what is a sentence? Problems of tokenization. MLTT Technical Report 4, Xerox.

Grus, J. (2015). *Data Science from Scratch: First Principles with Python*. O'Reilly Media, Sebastopol, California.

Guo, P. (2014). Python is now the most popular introductory teaching language at top U.S. universities. `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universit fulltext`. Retrieved November 23, 2015.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18.

Harris, R. and Taylor, T. J. (1997). *Landmarks in Linguistic Thought, The Western Tradition from Socrates to Saussure*. Routledge, London, 2nd edition.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, second edition edition.

Hausser, R. (2000). *Grundlagen der Computerlinguistik. Mensch-Maschine-Kommunikation in natürlicher Sprache*. Springer Verlag, Berlin Heidelberg New York.

Hausser, R. (2014). *Foundations of Computational Linguistics. Human-Computer Communication in Natural Language*. Springer Verlag, Berlin Heidelberg New York, 3rd edition.

Hoerl, A. E. (1962). Application of ridge analysis to regression problems. *Chemical Engineering Progress*, 58(3):54–59.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2007). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, 3rd edition.

Huang, J., Gao, J., Miao, J., Li, X., Wang, K., and Behr, F. (2010). Exploring web scale language models for search query processing. In *Proceedings of the 19th International World Wide Web Conference*, pages 451–460, Raleigh, North Carolina.

Huang, X., Acero, A., Alleva, F., Hwang, M.-Y., Jiang, L., and Mahaja, M. (1995). Microsoft highly intelligent speech recognizer: Whisper. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Detroit.

Imbs, P. and Quemada, B., editors (1971-1994). *Trésor de la langue française. Dictionnaire de la langue française du XIXe et du XXe siècle (1789-1960)*. Éditions du CNRS puis Gallimard, Paris. 16 volumes.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. Springer, New York.

Jelinek, F. (1990). Self-organized language modeling for speech recognition. In Waibel, A. and Lee, K.-F., editors, *Readings in Speech Recognition*. Morgan Kaufmann, San Mateo. Reprinted from an IBM Report, 1985.

Jelinek, F. (1997). *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, Massachusetts.

Jelinek, F. and Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In Gelsema, E. S. and Kanal, L. N., editors, *Pattern Recognition in Practice*, pages 38–397. North-Holland, Amsterdam.

Jensen, K., Heidorn, G., and Richardson, S., editors (1993). *Natural Language Processing: The PLNLP Approach*. Kluwer Academic Publishers, Boston.

Joachims, T. (2002). *Learning to Classify Text Using Support Vector Machines*. Kluwer Academic Publishers, Boston.

Jordan, C. (1874). Mémoire sur les formes bilinéaires. *Journal de mathématiques pures et appliquées, deuxième série*, 19:35–54.

Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing, An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson Education, Upper Saddle River, second edition.

Kaeding, F. W. (1897). *Häufigkeitswörterbuch der deutschen Sprache*. Selbstverlag des Herausgebers, Steglitz bei Berlin.

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

Kernighan, M. D., Church, K. W., and Gale, W. A. (1990). A spelling correction program based on a noisy channel model. In *Papers presented to the 13th International Conference on Computational Linguistics (COLING-90)*, volume II, pages 205–210, Helsinki.

Kiss, T. and Strunk, J. (2006). Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–525.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton.

Knuth, D. E. (1986). *The TeXbook*. Addison-Wesley, Reading, Massachusetts.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86.

Laplace, P. (1820). *Théorie analytique des probabilités*. Coursier, Paris, 3rd edition.

Legendre, A.-M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes*. Firmin Didot, Paris.

Levy, O., Goldberg, Y., and Dagan, I. (2015). Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225.

Lewis, D. D., Yang, Y., Rose, T. G., and Li, F. (2004). RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397.

Liang, P. (2005). Semi-supervised learning for natural language. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Linke, A., Nussbaumer, M., and Portmann, P. R. (2004). *Studienbuch Linguistik*. Niemeyer, Tübingen, 5. edition.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, Sebastopol, California, 5th edition.

Malmberg, B. (1983). *Analyse du langage au XXe siècle. Théorie et méthodes*. Presses universitaires de France, Paris.

Malmberg, B. (1991). *Histoire de la linguistique. De Sumer à Saussure*. Presses universitaires de France, Paris.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, Cambridge.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, Massachusetts.

Marcus, M., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Mast, M., Kummert, F., Ehrlich, U., Fink, G. A., Kuhn, T., Niemann, H., and Sagerer, G. (1994). A speech understanding and dialog system with a homogeneous linguistic knowledge base. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):179–194.

Mauldin, M. L. and Leavitt, J. R. R. (1994). Web-agent related research at the Center for Machine Translation. In *Proceedings of the ACM SIG on Networked Information Discovery and Retrieval*, McLean, Virginia.

McMahon, J. G. and Smith, F. J. (1996). Improving statistical language models performance with automatically generated word hierarchies. *Computational Linguistics*, 22(2):217–247.

Microsoft (2004). *Microsoft Office Word 2003 Rich Text Format (RTF) Specification*. Microsoft. RTF Version 1.8.

Mikheev, A. (2002). Periods, capitalized words, etc. *Computational Linguistics*, 28(3):289–318.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In *First International Conference on Learning Representations (ICLR2013), Workshop Proceedings*.

Mohri, M., Pereira, F. C. N., and Riley, M. (2000). The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32.

Moore, E. H. (1920). On the reciprocal of the general algebraic matrix, abstract. *Bull. Amer. Math. Soc*, 26:394–395.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, Massachusetts.

Nguyen, L., Abdou, S., Afify, M., Makhoul, J., Matsoukas, S., Schwartz, R., Xiang, B., Lamel, L., Gauvain, J.-L., Adda, G., Schwenk, H., and Lefevre, F. (2004). The 2004 BBN/LIMSI 10xRT English broadcast news transcription system. In *Proceedings DARPA RT04*, Palisades, New York.

Norvig, P. (2007). How to write a spelling corrector. `http://norvig.com/spell-correct.html`. Cited 30 November 2015.

Norvig, P. (2009). Natural language corpus data. In *Beautiful Data: The Stories Behind Elegant Data Solutions*, pages 219–242. O'Reilly Media, Sebastopol, California.

Och, F. J. (2012). Breaking down the language barrier – six years in. `http://googletranslate.blogspot.com`. Cited 4 June 2012.

Orwell, G. (1949). *Nineteen Eighty-Four*. Secker and Warburg, London.

Palmer, H. E. (1933). *Second Interim Report on English Collocations, Submitted to the Tenth Annual Conference of English Teachers under the Auspices of the Institute for Research in English Teaching*. Institute for Research in English Teaching, Tokyo.

Patterson, J. and Gibson, A. (2017). *Deep learning: a practitioner's approach*. O'Reilly, Sebastopol, California.

Payne, B. (2015). *Teach Your Kids to Code: A Parent-Friendly Guide to Python Programming*. No Starch Press, San Francisco.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Petruszewycz, M. (1973). L'histoire de la loi d'Estoup-Zipf: documents. *Mathématiques et Sciences Humaines*, 44:41–56.

Procter, P., editor (1995). *Cambridge International Dictionary of English*. Cambridge University Press, Cambridge.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Ray, E. T. (2003). *Learning XML*. O'Reilly, Sebastopol, California, 2nd edition.

Rayner, M., Carter, D., Bouillon, P., Digalakis, V., and Wirén, M., editors (2000). *The Spoken Language Translator*. Cambridge University Press, Cambridge.

Reynar, J. C. (1998). *Topic Segmentation: Algorithms and Applications*. PhD thesis, University of Pennsylvania, Philadelphia.

Reynar, J. C. and Ratnaparkhi, A. (1997). A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 16–19, Washington, DC.

Robins, R. H. (1997). *A Short History of Linguistics*. Longman, London, 4th edition.

Roche, E. and Schabes, Y., editors (1997). *Finite-State Language Processing*. MIT Press, Cambridge, Massachusetts.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rouse, R. H. and Rouse, M. A. (1974). The verbal concordance to the scriptures. *Archivum Fratrum Praedicatorum*, 44:5–30.

Sabah, G. (1990). *L'intelligence artificielle et le langage*. Hermès, Paris, 2nd edition.

Salton, G. (1988). *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Massachusetts.

Salton, G. and Buckley, C. (1987). Term weighting approaches in automatic text retrieval. Technical Report TR87-881, Department of Computer Science, Cornell University, Ithaca, New York.

Saporta, G. (2011). *Probabilités, analyse des données et statistiques*. Éditions Technip, Paris, 3rd edition.

Saussure, F. (1916). *Cours de linguistique générale*. Reprinted Payot, 1995, Paris.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:398–403 and 623–656.

Simone, R. (2007). *Fondamenti di linguistica*. Laterza, Bari, 10th edition.

Sinclair, J., editor (1987). *Collins COBUILD English Language Dictionary*. Collins, London.

Sorin, C., Jouvet, D., Gagnoulet, C., Dubois, D., Sadek, D., and Toularhoat, M. (1995). Operational and experimental French telecommunication services using CNET speech recognition and text-to-speech synthesis. *Speech Communication*, 17(3-4):273–286.

Spärck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21.

Stolcke, A. (2002). SRILM – an extensible language modeling toolkit. In *Proceedings International Conference Spoken Language Processing*, Denver.

Tesnière, L. (1966). *Éléments de syntaxe structurale*. Klincksieck, Paris, 2nd edition.

The Unicode Consortium (2012). *The Unicode Standard, Version 6.1 – Core Specification*. Unicode Consortium, Mountain View.

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

van Noord, G. and Gerdemann, D. (2001). An extendible regular expression compiler for finite-state approaches in natural language processing. In Boldt, O. and Jürgensen, H., editors, *Automata Implementation. 4th International Workshop on Implementing Automata, WIA'99, Potsdam, Germany, July 1999, Revised Papers*, volume 2214 of *Lecture Notes in Computer Science*, pages 122–139, Berlin Heidelberg New York. Springer Verlag.

van Rossum, G., Warsaw, B., and Coghlan, N. (2013). PEP 0008 – Style guide for Python code. `https://www.python.org/dev/peps/pep-0008/`. Retrieved November 23, 2015.

Verhulst, P.-F. (1838). Notice sur la loi que la population poursuit dans son accroissement. *Correspondance mathématique et physique*, 10:113–121.

Verhulst, P.-F. (1845). Recherches mathématiques sur la loi d'accroissement de la population. *Nouveaux Mémoires de l'Académie Royale des Sciences et Belles-Lettres de Bruxelles*, 18:1–42.

Wall, L., Christiansen, T., and Orwant, J. (2000). *Programming Perl*. O'Reilly, Sebastopol, California, 3rd edition.

Wang, K., Thrasher, C., Viegas, E., Li, X., and Hsu, B. (2010). An overview of Microsoft web n-gram corpus and applications. In *Proceedings of the NAACL HLT 2010: Demonstration Session*, pages 45–48, Los Angeles.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition.

Yu, H.-F., Ho, C.-H., Juan, Y.-C., and Lin, C.-J. (2013). LibShortText: A library for short-text classification and analysis. `http://www.csie.ntu.edu.tw/~cjlin/libshorttext`. Retrieved 1 November 2013.

Zampolli, A. (2003). Past and on-going trends in computational linguistics. *The ELRA Newsletter*, 8(3-4):6–16.

Zaragoza, H., Craswell, N., Taylor, M., Saria, S., and Robertson, S. (2004). Microsoft Cambridge at TREC-13: Web and HARD tracks. In *Proceedings of TREC-2004*, Gaithersburg.