

Cognitive Technologies

Pierre M. Nugues

# Python for Natural Language Processing

Programming with NumPy, scikit-learn,  
Keras, and PyTorch

*Third Edition*



# Cognitive Technologies

## **Editor-in-Chief**

Daniel Sonntag, German Research Center for AI, DFKI, Saarbrücken, Saarland, Germany

**Titles in this series now included in the Thomson Reuters Book Citation Index and Scopus!**

The Cognitive Technologies (CT) series is committed to the timely publishing of high-quality manuscripts that promote the development of cognitive technologies and systems on the basis of artificial intelligence, image processing and understanding, natural language processing, machine learning and human-computer interaction.

It brings together the latest developments in all areas of this multidisciplinary topic, ranging from theories and algorithms to various important applications. The intended readership includes research students and researchers in computer science, computer engineering, cognitive science, electrical engineering, data science and related fields seeking a convenient way to track the latest findings on the foundations, methodologies and key applications of cognitive technologies.

The series provides a publishing and communication platform for all cognitive technologies topics, including but not limited to these most recent examples:

- Interactive machine learning, interactive deep learning, machine teaching
- Explainability (XAI), transparency, robustness of AI and trustworthy AI
- Knowledge representation, automated reasoning, multiagent systems
- Common sense modelling, context-based interpretation, hybrid cognitive technologies
- Human-centered design, socio-technical systems, human-robot interaction, cognitive robotics
- Learning with small datasets, never-ending learning, metacognition and introspection
- Intelligent decision support systems, prediction systems and warning systems
- Special transfer topics such as CT for computational sustainability, CT in business applications and CT in mobile robotic systems

The series includes monographs, introductory and advanced textbooks, state-of-the-art collections, and handbooks. In addition, it supports publishing in Open Access mode.

Pierre M. Nugues

# Python for Natural Language Processing

Programming with NumPy, scikit-learn,  
Keras, and PyTorch

Third Edition



Springer

Pierre M. Nugues  
Department of Computer Science  
Lund University  
Lund, Sweden

ISSN 1611-2482  
Cognitive Technologies  
ISBN 978-3-031-57548-8  
<https://doi.org/10.1007/978-3-031-57549-5>

ISSN 2197-6635 (electronic)  
ISBN 978-3-031-57549-5 (eBook)

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2006, 2014, 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

If disposing of this product, please recycle the paper.

*À mes parents,  
À Madeleine*

# Preface to the Third Edition

Many things have changed since the last edition of this book. In all the areas of natural language processing, progress has been astonishing. Recent achievements in text generation even spurred a media interest that went beyond the traditional academic circles. Text processing has become more of a mainstream industrial tool, and countless companies now use it to some extent. A revision of this book was then necessary to adapt it to this recent evolution.

As in the two first editions, the intention is to expose the reader to the theories used in natural language processing but also to programming examples that are essential to a good understanding of the concepts. Although present in the previous editions, machine learning is even more pregnant and has replaced many of the earlier techniques to process text. Machine learning relies on mathematical principles that we cannot presuppose from a reader. New ways to combine the processing modules also appeared, and Python emerged as the dominant programming language in the field. This made a complete rewrite of the book even more indispensable to replace the programming language, present the mathematical background of machine learning, describe the new architectures, and update all the programming parts accordingly.

Along the chapters, the reader will get familiar with new topics, mathematical models, as well as programs to experiment with. Many new techniques build on the availability of text. Using Python notebooks, the reader will load small corpora, format the text, apply the models through the execution of pieces of code, discover gradually the theoretical parts, possibly modify the code or the parameters, and traverse theories and concrete problems through a constant interaction between the user and the machine. We tried to keep all the data sizes and hardware requirements reasonable so that a user can see instantly, or at least quickly, the results of most experiments on most machines.

All these programs are available in the form of Python notebooks from the GitHub repository: <https://github.com/pnugues/pnlp>. We hope the reader will enjoy them and that they will spark ideas to modify and improve the code. Notebooks are a wonderful tool that makes it so easy and engaging to test hypotheses. Ideally, they will inspire the reader to come up with new algorithms and, hopefully, results better

than ours. While this book is intended to be a textbook, contrary to the two previous editions, we did not include exercises at the end of chapters as we felt the notebooks were already exercises in themselves.

No preface goes without acknowledgments, and I would like to thank the PhD candidates I supervised between the two editions of this book, for all the stimulating discussions and work we did together; in chronological order: Peter Exner, Dennis Medved, Håkan Jonsson, and Marcus Klang. I would also like to thank Maj Stenmark and John Rausér Porsback for pointing out typos or mistakes in the last drafts, as well as Marcus again for advice on some programs.

A last word on me: August 15, 2016, was an unfortunate day in my life, when employees from a renovation company demolished the window of my office without proper warning, signaling, or securing the workplace. This left me with mutilated inner ears and a debilitating tinnitus. It considerably delayed this edition. Only the help of my wife, Charlotte, and the attention of my children, Andreas and Louise, helped me overcome this maddening condition. I hope one day research will find a way to cure it (Wang et al., [2019](#)).

Lund, Sweden  
January 2024

Pierre M. Nugues

# 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, Sweden  
April 2014

Pierre M. Nugues

# 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 book 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](http://www.perl.com)) or Prolog. Although I only tested the programs with SWI Prolog available from [www.swi-prolog.org](http://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, Sweden  
January 2006

Pierre M. Nugues

# Contents

<b>1</b>	<b>An Overview of Language Processing .....</b>	<b>1</b>
1.1	Applications of Language Processing.....	1
1.2	Evaluating the Applications .....	3
1.3	Why Speech and Language Processing Are Difficult .....	4
1.3.1	Ambiguity .....	4
1.3.2	Models and Their Implementation .....	5
1.4	The Domains We Will Cover .....	5
1.5	Further Reading .....	8
<b>2</b>	<b>A Tour of Python .....</b>	<b>9</b>
2.1	Why Python? .....	9
2.2	The Read, Evaluate, and Print Loop .....	10
2.3	Introductory Programs .....	11
2.4	Strings .....	11
2.4.1	String Index .....	12
2.4.2	String Operations and Functions .....	13
2.4.3	Slices .....	14
2.4.4	Special Characters .....	14
2.4.5	Formatting Strings .....	16
2.5	Data Identities and Types .....	16
2.6	Data Structures .....	17
2.6.1	Lists .....	17
2.6.2	List Copy .....	19
2.6.3	Built-in List Operations and Functions .....	20
2.6.4	Tuples .....	21
2.6.5	Sets .....	22
2.6.6	Built-in Set Functions .....	22
2.6.7	Dictionaries .....	23
2.6.8	Built-in Dictionary Functions .....	24
2.6.9	Counting the Letters of a Text .....	24

2.7	Control Structures .....	26
2.7.1	Conditionals .....	26
2.7.2	The <code>for</code> Loop .....	26
2.7.3	The <code>while</code> Loop .....	27
2.7.4	Exceptions .....	28
2.8	Functions .....	29
2.9	Documenting Functions .....	29
2.9.1	Docstrings .....	29
2.9.2	Type Annotations .....	30
2.10	Comprehensions and Generators .....	31
2.10.1	Comprehensions .....	31
2.10.2	Generators .....	32
2.10.3	Iterators .....	32
2.10.4	<code>zip</code> .....	33
2.11	Modules .....	34
2.12	Installing Modules .....	35
2.13	Basic File Input/Output .....	36
2.14	Collecting a Corpus from the Internet .....	37
2.15	Memo Functions and Decorators .....	38
2.15.1	Memo Functions .....	38
2.15.2	Decorators .....	39
2.16	Object-Oriented Programming .....	39
2.16.1	Classes and Objects .....	40
2.16.2	Subclassing .....	41
2.16.3	Counting with the Counter Class .....	42
2.17	Functional Programming .....	43
2.17.1	<code>map()</code> .....	43
2.17.2	Lambda Expressions .....	43
2.17.3	<code>reduce()</code> .....	44
2.17.4	<code>filter()</code> .....	44
2.18	Further Reading .....	45
3	<b>Corpus Processing Tools .....</b>	47
3.1	Corpora .....	47
3.1.1	Types of Corpora .....	48
3.1.2	Corpora and Lexicon Building .....	49
3.1.3	Corpora as Knowledge Sources .....	51
3.2	Finite-State Automata .....	52
3.2.1	A Description .....	52
3.2.2	Mathematical Definition of Finite-State Automata .....	53
3.2.3	Deterministic and Nondeterministic Automata .....	54
3.2.4	Building a Deterministic Automaton from a Nondeterministic One .....	54
3.2.5	Searching a String with a Finite-State Automaton .....	55
3.2.6	Operations on Finite-State Automata .....	57

3.3	Regular Expressions .....	59
3.3.1	Repetition Metacharacters .....	60
3.3.2	The Dot Metacharacter .....	61
3.3.3	The Escape Character .....	61
3.3.4	The Longest Match .....	61
3.3.5	Character Classes .....	63
3.3.6	Nonprintable Symbols or Positions .....	65
3.3.7	Union and Boolean Operators .....	65
3.3.8	Operator Combination and Precedence .....	66
3.4	Programming with Regular Expressions .....	67
3.4.1	Matching .....	67
3.4.2	A Simplified grep Program .....	68
3.4.3	Match Modifiers .....	69
3.4.4	Substitutions .....	70
3.4.5	Backreferences .....	70
3.4.6	Backreferences in the Pattern .....	71
3.4.7	Raw Strings .....	71
3.4.8	Python Escape Sequences .....	72
3.4.9	Backreferences and Substitutions .....	72
3.4.10	Match Objects .....	73
3.4.11	Parameterizable Regular Expressions .....	73
3.5	Finding Concordances .....	74
3.6	Lookahead and Lookbehind .....	76
3.7	Approximate String Matching .....	77
3.7.1	Edit Operations .....	77
3.7.2	Edit Operations for Spell Checking .....	78
3.7.3	Minimum Edit Distance .....	80
3.7.4	Computing the Minimum Edit Distance in Python .....	81
3.7.5	Searching Edits .....	82
3.8	Further Reading .....	82
4	<b>Encoding and Annotation Schemes .....</b>	85
4.1	Character Sets .....	86
4.1.1	Representing Characters .....	86
4.1.2	Unicode .....	87
4.1.3	Character Composition and Normalization .....	90
4.1.4	Unicode Character Properties .....	91
4.1.5	The Unicode Encoding Schemes .....	94
4.2	Locales and Word Order .....	95
4.2.1	Presenting Time, Numerical Information, and Ordered Words .....	95
4.2.2	The Unicode Collation Algorithm .....	97
4.2.3	Sorting with Python .....	98
4.3	Tabular Formats .....	100
4.3.1	The <code>csv</code> Module .....	100

4.3.2	Pandas .....	102
4.4	Markup Languages.....	103
4.4.1	An Outline of XML.....	103
4.4.2	XML and Databases .....	106
4.5	Collecting Corpora from the Web .....	106
4.5.1	Scraping Documents with Python.....	106
4.5.2	HTML .....	107
4.5.3	Parsing HTML .....	107
4.6	Further Reading .....	109
<b>5</b>	<b>Python for Numerical Computations .....</b>	<b>111</b>
5.1	Dataset.....	112
5.2	Vectors .....	112
5.2.1	Representing the Counts .....	113
5.2.2	Data Types .....	113
5.2.3	Size of the Vectors .....	113
5.2.4	Indices and Slices .....	114
5.2.5	Operations .....	114
5.2.6	Comparison with Lists .....	114
5.2.7	PyTorch.....	115
5.2.8	Mathematical Background: The Vector Space .....	117
5.3	NumPy Functions .....	117
5.4	The Dot Product.....	118
5.4.1	Euclidian Norm of a Vector .....	119
5.4.2	Cosine of Two Vectors .....	119
5.4.3	The Dot Product in Mathematics.....	120
5.4.4	Elementwise Product .....	120
5.5	Matrices .....	121
5.5.1	Matrices in NumPy .....	121
5.5.2	Indices and Slices .....	121
5.5.3	Order and Dimensions of a Tensor.....	122
5.5.4	Addition and Multiplication by a Scalar .....	122
5.5.5	Matrices in PyTorch .....	123
5.5.6	Matrix Creation Functions .....	124
5.5.7	Applying Functions .....	124
5.5.8	Transposing and Reshaping Arrays .....	125
5.5.9	Reshaping with PyTorch .....	126
5.5.10	Broadcasting .....	127
5.6	Matrix Products .....	128
5.6.1	Matrix-Vector Multiplication.....	128
5.6.2	Matrix Multiplication .....	129
5.6.3	Computing the Cosines .....	129
5.7	Elementary Mathematical Background for Matrices .....	131
5.7.1	Linear and Affine Maps .....	131
5.7.2	Linear Functions and Vectors .....	131

5.7.3	Matrix Example .....	132
5.7.4	Transpose .....	132
5.7.5	Matrices and Rotations .....	133
5.7.6	Function Composition .....	134
5.7.7	Application to Rotations .....	134
5.7.8	Inverse Function .....	134
5.7.9	Inverting a Matrix in NumPy and PyTorch .....	135
5.8	Application to Neural Networks .....	135
5.8.1	Matrices and Datasets .....	136
5.8.2	Matrices and PyTorch: One Layer .....	137
5.8.3	More Layers .....	138
5.9	Automatic Differentiation .....	138
5.10	Further Reading .....	139
<b>6</b>	<b>Topics in Information Theory and Machine Learning .....</b>	<b>141</b>
6.1	Codes and Information Theory .....	141
6.1.1	Entropy .....	142
6.1.2	Python Implementation .....	143
6.1.3	Huffman Coding .....	144
6.1.4	Cross-Entropy .....	148
6.1.5	Perplexity and Cross-Perplexity .....	149
6.2	Entropy and Decision Trees .....	149
6.2.1	A Toy Dataset .....	150
6.2.2	Decision Trees .....	151
6.2.3	Inducing Decision Trees Automatically .....	152
6.2.4	Numerical Attributes .....	154
6.3	Encoding Categorical Values as Numerical Features .....	154
6.4	Programming: Inducing Decision Trees with Scikit-Learn .....	156
6.4.1	Conversion of Categorical Data .....	156
6.4.2	Inducing a Decision Tree with Scikit-Learn .....	158
6.4.3	Evaluating a Model .....	158
6.5	Further Reading .....	159
<b>7</b>	<b>Linear and Logistic Regression .....</b>	<b>161</b>
7.1	Linear Classifiers .....	161
7.2	Choosing a Dataset .....	162
7.3	Linear Regression .....	163
7.3.1	Least Squares .....	164
7.3.2	Least Absolute Deviation .....	165
7.4	Notations in an $n$ -Dimensional Space .....	165
7.5	Gradient Descent .....	167
7.5.1	Mathematical Description .....	168
7.5.2	Gradient Descent and Linear Regression .....	169
7.6	Regularization .....	171
7.6.1	The Analytical Solution Again .....	171
7.6.2	Inverting $X^T X$ .....	172

7.6.3	Regularization .....	172
7.7	Linear Classification .....	173
7.7.1	An Example .....	173
7.7.2	Classification in an $N$ -Dimensional Space .....	176
7.7.3	Linear Separability .....	176
7.7.4	Classification vs. Regression .....	177
7.8	Perceptron .....	178
7.8.1	The Heaviside Function .....	178
7.8.2	The Iteration .....	179
7.8.3	The Two-Dimensional Case .....	179
7.8.4	Stop Conditions .....	180
7.9	Logistic Regression .....	180
7.9.1	Fitting the Weight Vector .....	181
7.9.2	Gradient Ascent ... .....	182
7.10	Gradient Descent Optimization .....	185
7.10.1	The Momentum .....	186
7.10.2	RMSprop .....	186
7.11	Programming Logistic Regression with Scikit-Learn .....	187
7.11.1	Representing the Dataset .....	187
7.11.2	Scaling the Data .....	188
7.11.3	Loading the Dataset from a File .....	188
7.11.4	Fitting a Model with Scikit-Learn .....	189
7.11.5	The Logistic Regression Model .....	190
7.11.6	The Loss .....	190
7.11.7	Comparing Cross Entropy and the Squared Error .....	191
7.11.8	Multinomial Logistic Regression .....	192
7.12	Evaluation of Classification Systems .....	193
7.12.1	Accuracy .....	193
7.12.2	Precision and Recall .....	194
7.13	Further Reading .....	195
8	<b>Neural Networks .....</b>	197
8.1	Representation and Notation .....	197
8.2	Feed-Forward Computation .....	198
8.2.1	The Perceptron and Logistic Regression .....	200
8.2.2	Hidden Layers .....	200
8.3	Backpropagation .....	201
8.3.1	Presentation .....	201
8.3.2	Naive Gradient Descent .....	202
8.3.3	Breaking Down the Computation .....	203
8.3.4	Gradient with Respect to the Input .....	203
8.3.5	Gradient with Respect to the Weights .....	206
8.4	Applying Neural Networks to Datasets .....	207
8.5	Programming Neural Networks .....	208
8.5.1	Data Representation and Preprocessing .....	209

8.5.2	Keras .....	210
8.5.3	PyTorch.....	213
8.6	Classification with More than Two Classes.....	219
8.6.1	Cross Entropy Loss .....	220
8.6.2	The Softmax Function .....	221
8.7	Multiclass Classification with Keras .....	222
8.8	Multiclass Classification with PyTorch .....	224
8.9	Backpropagation in PyTorch .....	226
8.10	Further Reading .....	228
<b>9</b>	<b>Counting and Indexing Words .....</b>	<b>229</b>
9.1	Text Segmentation .....	229
9.1.1	What Is a Word? .....	229
9.1.2	Breaking a Text into Words and Sentences .....	231
9.2	Tokenizing Words .....	232
9.2.1	Defining Content.....	232
9.2.2	Using Boundaries .....	233
9.2.3	Improving Tokenization .....	234
9.2.4	Tokenizing Using Classifiers .....	235
9.3	Sentence Segmentation .....	236
9.3.1	The Ambiguity of the Period Sign .....	236
9.3.2	Rules To Disambiguate the Period Sign .....	236
9.3.3	Using Regular Expressions .....	237
9.3.4	Improving the Segmenter Using Lexicons .....	237
9.3.5	Sentence Detection Using Classifiers .....	238
9.4	Word Counting .....	239
9.4.1	Some Definitions.....	239
9.4.2	Counting Words with Python.....	239
9.4.3	The Counter Class .....	241
9.4.4	A Crash Program To Count Words with Unix .....	242
9.5	Retrieval and Ranking of Documents .....	243
9.5.1	Document Indexing .....	243
9.5.2	Building an Inverted Index in Python .....	244
9.5.3	Representing Documents as Vectors .....	246
9.5.4	Vector Coordinates.....	247
9.5.5	Ranking Documents .....	248
9.6	Categorizing Text .....	249
9.6.1	Corpora .....	249
9.6.2	Building a Categorizer with Scikit-Learn.....	249
9.7	Further Reading .....	251
<b>10</b>	<b>Word Sequences .....</b>	<b>253</b>
10.1	Modeling Word Sequences .....	253
10.2	<i>N</i> -Grams .....	254
10.2.1	Counting Bigrams with Python .....	255
10.2.2	Counting Bigrams with Unix .....	255

10.3	Probabilistic Models of a Word Sequence .....	256
10.3.1	The Maximum Likelihood Estimation .....	256
10.3.2	Using ML Estimates with <i>Nineteen Eighty-Four</i> .....	258
10.4	Smoothing $N$ -Gram Probabilities .....	261
10.4.1	Sparse Data .....	261
10.4.2	Laplace's Rule .....	261
10.4.3	Good–Turing Estimation .....	263
10.5	Using $N$ -Grams of Variable Length .....	265
10.5.1	Linear Interpolation .....	266
10.5.2	Back-Off .....	267
10.5.3	Katz's Back-Off Model .....	268
10.5.4	Kneser–Ney Smoothing Model .....	269
10.6	Industrial $N$ -Grams .....	270
10.7	Quality of a Language Model .....	270
10.7.1	Intuitive Presentation .....	270
10.7.2	Entropy Rate .....	271
10.7.3	Cross Entropy .....	271
10.7.4	Perplexity .....	272
10.8	Generating Text from a Language Model .....	272
10.8.1	Using the Multinomial Distribution .....	273
10.8.2	Transforming the Distribution .....	275
10.9	Collocations .....	276
10.9.1	Word Preference Measurements .....	276
10.9.2	Extracting Collocations with Python .....	280
10.9.3	Applying Collocation Measures .....	282
10.10	Further Reading .....	283
11	<b>Dense Vector Representations</b> .....	285
11.1	Vector Representations .....	285
11.2	Dimensionality Reduction .....	286
11.2.1	Singular Value Decomposition .....	286
11.2.2	Data Representation and Preprocessing .....	288
11.2.3	Computing a Singular Value Decomposition .....	288
11.3	Applying a SVD to the <i>Salammbo</i> Dataset .....	288
11.3.1	Counts of Letter A .....	289
11.3.2	Counts of all the Characters .....	291
11.3.3	The Characters in a Space of Documents .....	291
11.3.4	Singular Value Decomposition and Principal Component Analysis .....	292
11.4	Latent Semantic Indexing .....	293
11.5	Word Embeddings from a Cooccurrence Matrix .....	294
11.5.1	Preprocessing .....	295
11.5.2	Counting the Cooccurrences .....	295
11.5.3	Applying a PCA .....	296
11.5.4	Saving the Vectors .....	297

11.6	Embeddings' Similarity .....	297
11.6.1	Cosine Similarity .....	297
11.6.2	Programming .....	298
11.7	From Cooccurrences to Mutual Information .....	299
11.8	GloVe .....	300
11.8.1	Model .....	300
11.8.2	Loss .....	301
11.8.3	Embeddings .....	302
11.8.4	The Dataloader .....	302
11.8.5	Programming the Model .....	303
11.8.6	Computing the Embeddings .....	304
11.8.7	Semantic Similarity .....	304
11.9	Word Embeddings from Neural Networks .....	304
11.9.1	word2vec .....	304
11.9.2	CBOW Architecture .....	305
11.9.3	Programming CBOW .....	306
11.9.4	Skipgrams .....	309
11.10	Application of Embeddings to Language Detection .....	313
11.10.1	Corpus .....	314
11.10.2	Preprocessing and Balancing the Dataset .....	315
11.10.3	Character $N$ -Grams .....	317
11.10.4	Encoding the $N$ -Grams .....	317
11.10.5	Building $X$ and $y$ .....	318
11.10.6	Bags of Embeddings .....	319
11.10.7	Model .....	320
11.10.8	Training Loop .....	321
11.10.9	Evaluation .....	322
11.11	Further Reading .....	322
12	Words, Parts of Speech, and Morphology .....	325
12.1	Words .....	325
12.1.1	Parts of Speech .....	325
12.1.2	Grammatical Features .....	326
12.1.3	Two Significant Parts of Speech: The Noun and the Verb .....	327
12.2	Standardized Part-of-Speech Tagsets and Grammatical Features .....	329
12.2.1	Multilingual Part-of-Speech Tags .....	329
12.2.2	Multilingual Grammatical Features .....	331
12.3	The CoNLL Format .....	333
12.4	A CoNLL Reader in Python .....	334
12.5	Lexicons .....	337
12.5.1	Encoding a Dictionary .....	339
12.6	Morphology .....	340
12.6.1	Morphemes .....	340

12.6.2	Morphs .....	342
12.6.3	Inflection and Derivation .....	342
12.6.4	Language Differences .....	346
12.7	Morphological Parsing .....	347
12.7.1	Two-Level Model of Morphology .....	347
12.7.2	Interpreting the Morphs .....	347
12.7.3	Finite-State Transducers .....	349
12.7.4	Conjugating a French Verb .....	349
12.7.5	Application to Romance Languages .....	351
12.7.6	Ambiguity .....	351
12.7.7	Operations on Finite-State Transducers .....	352
12.8	Further Reading .....	353
13	<b>Subword Segmentation .....</b>	355
13.1	Deriving Morphemes Automatically .....	355
13.2	Byte-Pair Encoding .....	357
13.2.1	Outline of the Algorithm .....	357
13.2.2	Pretokenization .....	357
13.2.3	The Initial Vocabulary .....	358
13.2.4	Counting the Bigrams .....	359
13.2.5	Merging Pairs .....	360
13.2.6	Constructing the Merge Rules .....	360
13.2.7	Encoding .....	361
13.2.8	Tokenizing .....	362
13.2.9	Visualizing the Whitespace .....	362
13.2.10	Using Bytes .....	363
13.3	The WordPiece Tokenizer .....	364
13.3.1	Pretokenization and Initial Vocabulary .....	364
13.3.2	Computing the Gains .....	365
13.3.3	Constructing the Subwords .....	366
13.3.4	Encoding .....	367
13.3.5	Tokenization .....	368
13.3.6	BERT's WordPiece .....	368
13.4	Unigram Tokenizer .....	369
13.4.1	Initial Class .....	370
13.4.2	Estimating the Probabilities .....	371
13.4.3	Finding the Best Segmentation .....	372
13.4.4	Expectation-Maximization (EM) .....	375
13.4.5	Creating the Vocabulary .....	376
13.5	The SentencePiece Tokenizer .....	377
13.6	Hugging Face Tokenizers .....	377
13.6.1	Pretrained BPE .....	377
13.6.2	Training BPE .....	378
13.7	Further Reading .....	379

<b>14</b>	<b>Part-of-Speech and Sequence Annotation</b>	381
14.1	Resolving Part-of-Speech Ambiguity	381
14.2	Baseline	383
14.3	Evaluation	385
14.4	Part-of-Speech Tagging with Linear Classifiers	385
14.5	Programming a Part-of-Speech Tagger with Logistic Regression	388
14.5.1	Building the Feature Vectors	389
14.5.2	Encoding the Features	390
14.5.3	Applying the Classifier	391
14.6	Part-of-Speech Tagging with Feed-Forward Networks	391
14.6.1	Programming a Single Layer Network for POS Tagging	391
14.6.2	Training the Model	393
14.6.3	Networks with Hidden Layers	394
14.7	Embeddings	395
14.8	Recurrent Neural Networks	397
14.8.1	Programming RNNs for POS Tagging	399
14.8.2	Dropout	405
14.8.3	LSTM	406
14.8.4	POS Tagging with LSTMs	408
14.8.5	Further Improvements	409
14.9	Named Entity Recognition and Chunking	409
14.9.1	Noun Groups and Verb Groups	409
14.9.2	Named Entities	410
14.10	Group Annotation Using Tags	411
14.10.1	The IOB Tagset	411
14.10.2	The IOB2 Tagset	413
14.10.3	The BIOES Tagset	414
14.10.4	Extending BIO to Two or More Groups	414
14.10.5	Annotation Examples from CoNLL 2000, 2002, and 2003	414
14.11	Recurrent Networks for Named Entity Recognition	416
14.12	Conditional Random Fields	417
14.12.1	Modeling the $y$ Transitions	418
14.12.2	Adding the $x$ Input	419
14.12.3	Predicting a Sequence	420
14.12.4	Adding a CRF Layer to a LSTM Network	420
14.13	Tokenization	421
14.14	Further Reading	422
<b>15</b>	<b>Self-Attention and Transformers</b>	425
15.1	The Transformer Architecture	425
15.2	Self-Attention	426
15.2.1	GloVe Static Embeddings	427

15.2.2	Using Weighted Cosines to Create Contextual Embeddings .....	428
15.2.3	Dot-Product Attention .....	429
15.2.4	Projecting the Input .....	431
15.2.5	Programming the Attention Function .....	431
15.2.6	Adding the Query, Key, and Value Matrices .....	433
15.3	Multihead Attention .....	433
15.4	Residual Connections .....	435
15.5	Feedforward Sublayer .....	437
15.6	The Encoder Layers .....	437
15.7	The Complete Encoder .....	438
15.8	Input Embeddings .....	439
15.9	Positional Encodings .....	439
15.10	Converting the Input Words .....	440
15.11	Improving the Encoder .....	442
15.12	PyTorch Transformer Modules .....	443
15.12.1	The PyTorch Multihead Attention Class .....	443
15.12.2	The PyTorch TransformerEncoder Class .....	444
15.13	Application: Sequence Annotation .....	444
15.14	Application: Classification .....	446
15.15	Further Reading .....	446
<b>16</b>	<b>Pretraining an Encoder: The BERT Language Model .....</b>	<b>449</b>
16.1	Pretraining Tasks .....	450
16.2	Creating the Dataset .....	451
16.2.1	Input Sequence .....	451
16.2.2	Next Sentence Prediction .....	452
16.2.3	Masked Language Model .....	453
16.3	The Input Embeddings .....	454
16.4	Creating a Batch .....	456
16.5	BERT Architecture .....	457
16.6	BERT Pretraining .....	458
16.7	The Training Loop .....	460
16.8	BERT Applications .....	460
16.8.1	Classification .....	460
16.8.2	Sequence Annotation .....	461
16.8.3	Question Answering .....	463
16.8.4	Question Answering Systems .....	464
16.9	Using a Pretrained Model .....	465
16.10	The IMDB Dataset .....	466
16.11	Tokenization .....	467
16.12	Fine-Tuning .....	468
16.12.1	Architecture .....	468
16.12.2	Training Parameters .....	469
16.12.3	Training .....	469

16.12.4	Prediction.....	470
16.12.5	Freezing Layers .....	470
16.13	Further Reading .....	472
<b>17</b>	<b>Sequence-to-Sequence Architectures: Encoder-Decoders and Decoders .....</b>	<b>473</b>
17.1	Parallel Corpora .....	473
17.2	Alignment .....	474
17.3	The Encoder-Decoder Architecture .....	476
17.4	Encoder-Decoder Transformers .....	478
17.4.1	Masked Attention .....	478
17.4.2	Cross-Attention .....	480
17.4.3	Evaluating Translation .....	481
17.5	Programming: Machine Translation .....	482
17.5.1	The PyTorch TransformerDecoder Class .....	482
17.5.2	The PyTorch Transformer Class .....	483
17.5.3	Preparing the Dataset .....	484
17.5.4	Indexing the Symbols .....	485
17.5.5	Building the Transformer Model .....	486
17.5.6	Configuring the Model and Training Procedure .....	488
17.5.7	Datasets and Dataloaders .....	489
17.5.8	Training the Model .....	490
17.5.9	Decoding a Sentence .....	492
17.5.10	Improving the Performance .....	494
17.6	Decoders .....	495
17.6.1	Decoder Architecture .....	496
17.6.2	Training Procedure .....	497
17.7	Further Developments .....	499
<b>References</b> .....	<b>501</b>	
<b>Index</b> .....	<b>513</b>	

# Chapter 1

## An Overview of Language Processing



Γνῶθι σεαυτόν

'Know thyself'

Inscription at the entrance to Apollo's Temple at Delphi

Language is the embodiment of the cognitive capacities of human beings. We use it every day in conversations by listening and talking, or by reading and writing. It is probably our preferred mode of communication and interaction. So unattainable it can be, modeling, replicating, and mechanizing this human faculty have been a dream from the beginnings of computer science. Ideally, natural 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 immense technical challenges. However, although perfection has not been reached yet, 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 understanding techniques have already achieved results ranging from very promising to near-perfect. The description of these techniques is the subject of this book.

### 1.1 Applications of Language Processing

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, but they gradually improved and have now become industrial products.

The internet is certainly the main driver of these advances. It made multilingual text available like never before and provided an easy access to this raw material of natural language processing. It also created new needs such as the translation of text and information from any language in any other language. This daunting quantity of data spurred the creation of new tools, mostly based on statistics and machine learning. One the major trend of it is the “mathematization” of the field that went from ad-hoc logical rules to techniques dominated by algebra and calculus.

Mathematical theories or tools can be difficult to understand. Their abstract nature makes them intimidating to many. To appreciate a mathematical model, it is often easier to associate it with a concrete problem as these application examples:

**Spelling and grammar checkers.** These programs are now ubiquitous in text processors, and hundred of millions of people use them every day. Spelling checkers have been based primarily on computerized dictionaries. They can now 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 logical rules or mathematical language models to detect common grammar and style errors.

**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 or related concepts. Using statistics on words, popularity measures, and user interactions, text retrieval systems rank the documents and present their results instantly to the user.

**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 *Speech Recognition* or Google’s *Voice Search* have high performance and recognize many languages. Some systems transcribe radio, TV broadcast news, and even songs providing thus automatic subtitles.

**Voice control of domestic devices.** These systems are embedded in objects to provide them with a friendlier interface and when your hands are busy like when driving cars. 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. One challenge they still have to overcome is to operate in noisy environments that impair the 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. 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.

**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. Question answering systems have now improved to the point that they match or surpass human performance.

**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 130 languages and is used by more than 600 million people every day.

## 1.2 Evaluating the Applications

Unlike other computer programs, the results of language processing techniques rarely hit a 100% success rate. Their performance is merely assessed in statistical terms. Language processing techniques become mature and usable when they operate above a certain precision and at an acceptable cost.

All the applications we have listed have seen a rapid progress. However, this was sometimes hard to quantify accurately as the metrics and evaluation conditions could be different. This prompted the creation of multiple evaluation sites and benchmarking suites, where given a task, users can upload their results and immediately see their score with a unified metric. Notable evaluation benchmarks include GLUE (Wang et al. 2018) in text classification, SQuAD (Rajpurkar et al. 2016, 2018) for questions answering, and Kaggle<sup>1</sup> for general machine learning.

Competitors usually show on a leaderboard that sometimes changes everyday. This makes it difficult to track the state of the art as it may change overnight; we refer to these sites instead. Other more traditional evaluations include the Workshops

---

<sup>1</sup> <https://www.kaggle.com/competitions>.

on Machine Translation (WMT) and the shared tasks of the Conference on Natural Language Learning (CoNLL).

## 1.3 Why Speech and Language Processing Are Difficult

So far, for the applications we mentioned, machines 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.

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

The boys eat the sandwiches.

Using the International Phonetic Association (IPA) symbols, a perfect phonemic transcription of this utterance would yield the transcription:

[ˈðəb'ɔrz'i:t̬ ðəs'ændwidʒɪz],

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.

Historically, a way to resolve ambiguity has been to use a conjunction of lexical, syntactic, and semantic 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.

Current systems would now use mathematical models of word sequences and contexts instead. Such models on word occurrences drawn from large quantities of

texts—corpora—can capture grammatical and semantic patterns, as well as common sense. Improbable alternatives <boys eat sand> and <buoys eat sand> are highly unlikely in corpora and modern language models would rule them out.

### 1.3.2 *Models and Their Implementation*

Natural language processing often 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 such models also involve the study of the human mind. This gives a measure of the complexity of the description and the representation of language.

The NLP field has seen a lot of theories and models happen. 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 current or potential models require massive computing power. Processors and storage able to support their implementation with substantial corpora, for instance, were not widely available until recently.

However, in the last two decades models have matured, data has become available, and computing power has become inexpensive. Although models and implementations are rarely perfect, they now enable us to obtain exceptional results, unimaginable a few years ago. Most use a limited set of techniques pertaining to the theory of probability, statistics, and machine learning that we will consider throughout this book.

## 1.4 The Domains We Will Cover

In this book, we will cover recent models and architectures that have been instrumental in the recent progress of natural language processing. Each chapter is dedicated to a specific topic that we will relate to applications and illustrate with Python programs. As interaction is of primary importance to analyze the behavior of an algorithm or a piece of code, the programs are all available in the form of notebooks where readers can run the different pieces, modify them, see the results, and ultimately understand them.

This book does not presuppose a deep knowledge of Python. Chapter 2, *A Tour of Python*, gives an introduction to this language, especially directed to text processing. This will enable us to touch all the data structures and programming constructs we need to implement our algorithms.

Large collections of texts form the raw material of natural language processing. Chapter 3, *Corpus Processing Tools*, as a follow up to the Python introduction,

describes regular expressions, both a simple and efficient tool to process text. We will see how to use them in Python to match patterns in text and apply some transformations. This chapter also reviews techniques to carry out approximate string matching.

The internet is multilingual and languages use different scripts. Chapter 4, *Encoding and Annotation Schemes*, describes Unicode, a standard to encode about all the existing characters, and the Unicode regular expressions. Once encoded, most texts not only contain sequences of characters, but also embed a structure in the form of markups. This chapter outlines them as well as elementary techniques to collect corpora from the internet and parse their markup.

Machine learning is the main technique we use in this book. Chapter 5, *Python for Numerical Computations*, describes NumPy arrays and PyTorch tensors, the fundamental structures to represent and process numerical data in Python. This chapter also provides a reminder on the mathematical operations on vectors and matrices and how to apply them with Python.

Chapter 6, *Topics in Information Theory and Machine Learning*, proceeds with elementary concepts of information theory such as entropy and perplexity. Using these concepts, a small dataset, and the scikit-learn machine-learning toolkit, we will create our first classifier based on decision trees.

Logistic regression is a linear classification technique and a fundamental element of many machine-learning systems. Chapter 7, *Linear and Logistic Regression*, introduces it as well as gradient descent, a technique to fit the parameters of a logistic regression model to a dataset. Using scikit-learn again, we will train a logistic regression classifier to determine the language of a text from its character counts.

Logistic regression is an effective, but elementary technique. Chapter 8, *Neural Networks*, describes how we can extend it by stacking more layers and functions, and create a feed-forward neural network. This chapter also describes the backpropagation algorithm to adapt gradient descent to networks with multiple layers. Using Keras and PyTorch this time, we will train new neural networks to classify our texts.

In the chapters so far, we did not use the concept of word. Chapter 9, *Counting and Indexing Words*, describes how to segment a text in words and sentences either using regular expressions or a classifier. It also explains how to count the words of a text and index all the words contained in a collection of texts. Finally, it outlines a document representation using vectors of word frequencies and applies this representation in a new PyTorch model to classify texts.

Starting from the words counts of the previous chapter, Chap. 10, *Word Sequences*, introduces words sequences,  $N$ -grams, and language models. It illustrates them with programs to derive sequence probabilities and smooth their distributions. We finally use these probabilities to generate text sequences automatically.

In the two previous chapters, we represented the words as strings or Boolean vectors. Chapter 11, *Dense Vector Representations*, describes techniques to convert the words into relatively small numerical vectors also called dense representations. We examine more specifically principal component analysis as well as neural network alternatives, GloVe and CBOW. The chapter contains programs to experiment with

these techniques as well as a new kind of classifier using dense representations to replace the word or character input.

The focus of this book is mostly on machine-learning techniques. Chapter 12, *Words, Parts of Speech, and Morphology*, leaves this field a bit and describes how to analyze the words with traditional grammatical concepts. It introduces the parts of speech i.e. the linguistic categories that we can assign to words with similar syntactic or semantic properties. It presents the major annotation standards and how to decompose a word into syntactic or semantic elements through a morphological analysis.

Starting from a word decomposition that uses morphemes, Chap. 13, *Subword Segmentation*, describes purely statistical methods. These methods use counts of pairs of characters and language models to derive limited vocabularies of subwords from large corpora. This chapter contains programs that implement the byte-pair encoding, WordPiece, and unigram algorithms.

Chapter 14, *Part-of-Speech and Sequence Annotation*, describes a selection of neural network architectures to annotate the words of a text with their part of speech. Starting from feed-forward networks, we will train models on annotated datasets and apply them to a text. We will move on to recurrent networks that fit well the sequential nature of sentences. We will illustrate each architecture with a program. As application, we will consider the annotation with parts of speech and then the recognition of names in sentences, either people, countries, or organizations.

After feed-forward and recurrent, the transformer architecture is a third kind of network. Chapter 15, *Self-Attention and Transformers*, dissects the transformer architecture starting from the core concept of attention. It outlines its building blocks with small programs in PyTorch so that the reader can experiment with them. The chapter concludes with an implementation of a sequence annotation and a classifier with the encoder part of a transformer.

Transformers have the capacity to store a huge quantity of information. Chapter 16, *Pretraining an Encoder: The BERT Language Model*, describes how we can pretrain the encoder part of a transformer on very large raw corpora to fit their parameters from word associations. It then describes how to fine-tune the parameters for applications such as classification, sequence annotation, and question answering.

The transformer consisted originally of two parts: the encoder and the decoder. Chapter 17, *Sequence-to-Sequence Architectures: Encoder-Decoders and Decoders*, describes this complete structure and, to make it concrete, illustrates it with a machine-translation program in PyTorch. For sake of feasibility, we replace the word input with characters so that readers can train the model on a simple laptop. It produces, nonetheless, surprising good results given its simplicity. The chapter concludes with a description of the decoder alone and how we can apply it to text generation, dialogue, and question answering.

## 1.5 Further Reading

The internet has become the main source of pedagogical and technical references: on-line courses, digital libraries, general references, corpus, and software resources, together with registries and portals. Wikipedia contains definitions and general articles on concepts and theories used in machine learning and natural language processing.

Many programs are available as open source. They include speech synthesis and recognition, language models, morphological analysis, parsing, transformers, and so on. The spaCy library<sup>2</sup> is an open-source NLP toolkit in Python that features many components to train and apply models. The Natural Language Toolkit (NLTK)<sup>3</sup> is another valuable suite of open-source Python programs, datasets, and tutorials. It has a companion book: *Natural Language Processing with Python* by Bird et al. (2009). On the machine-learning side, the scikit-learn toolkit<sup>4</sup> (Pedregosa et al. 2011) is an easy-to-use, general purpose set of machine-learning algorithms. It has excellent documentation and tutorials.

The source code of these three toolkits, spaCy, NLTK, and scikit-learn, is available from GitHub and is worth examining, at least partly. To understand how a program is designed; how its files are structured; what are the best coding practices, reading is code is always informative. Beyond these toolkits, GitHub has an amazing wealth of open-source code and, when in need to understand an algorithm or a model, searching GitHub may help you find the rare gem.

As the field is constantly evolving, reading scientific papers is necessary to keep up with the changes. A starting point is the ACL anthology,<sup>5</sup> an extremely valuable source of research papers (journal and conferences) published under the auspices of the ACL. Such papers are generally reliable as they are reviewed by other scientists before they are published.

A key difference with scientific publishing from a few years ago is the speed of dissemination and the emergence of a permanent benchmarking. Many authors, as soon as they come with a new finding or surpass the state of the art in a specific application, publish a paper immediately and submit it to a conference later. arXiv<sup>6</sup> is the major open-access repository of such papers.

---

<sup>2</sup> <https://spacy.io/>.

<sup>3</sup> <https://www.nltk.org/>.

<sup>4</sup> <https://scikit-learn.org/>.

<sup>5</sup> <https://aclanthology.org/>.

<sup>6</sup> <https://arxiv.org/>.

# Chapter 2

## A Tour of Python



学 知 见 闻 不  
至于之之之之闻不  
行不若若若若闻不  
之而行知见之见之  
止矣

Chinese tenet sometimes attributed to Xunzi, usually abridged in English as  
*I hear and I forget, I see and I remember, I do and I understand.*

### 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 ←————— We create text and assign it with a string
>>> text = 'Result:' ←————— And we print both text and c
>>> print(text, 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')
```

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')
```

Figure 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_opening = """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_opening` 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

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

Fig. 2.1 Summary of the main Python operators

wrap the line so that it fits our text editor, we will use the backslash continuation character, \, as in:

```
iliad_opening2 = 'Sing, O goddess, the anger of Achilles son of \
Peleus, that brought countless ills upon the Achaeans.'
```

where the line break is ignored and `iliad_opening2` 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 a string 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 = 'éèëëë'  
accented_e.upper()      # 'ÉÈËËË'  
accented_E = 'ÉÈËËË'  
accented_E.lower()      # 'éèëëë'
```

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 c in iliad_opening:  
    if c in 'aeiou':  
        text_vowels = text_vowels + c  
print(text_vowels)  # 'ioeeaeoieooeeuaououeiuoaeaea'
```

We can abridge the statement:

```
text_vowels = text_vowels + c
```

into

```
text_vowels += c
```

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:]
```

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.

**Table 2.1** 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	\u000000152	Unicode code point, 32 bits, here œ
\a	Bell		
\'	Single quote		
\"	Double quote		
\\\	Backslash		

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 2.1 shows a list of escape sequences with their meaning.

The right column in Table 2.1 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 œ}, or its code point, \u00D6 and \u0152. We review both the ASCII and Unicode schemes in Chap. 4, *Encoding and Annotation Schemes*. We can also use \ooo octal and \xhh hexadecimal sequences:

```
'\N{COMMERCIAL AT}'      # 'œ'
'\x40'                   # 'œ'
'\100'                   # 'œ'
'\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. 3.4.

### 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'
'{0} string {1}'.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 Identities and Types

Python uses objects to model data. This means that all the variables we have used so far refer to objects. Each Python object has an identity, a value, and a type to categorize it. The identity is a unique number that is assigned to the object at runtime when it is created so that there is no ambiguity on it. The type defines what kind of operations we can apply to this object.

We return the object identity with the `id()` function and its type with `type()`. The respective value, identity, and type of 12 are:

```
12          # 12
id(12)      # 140390267284112
type(12)    # <class 'int'>
```

When we assign the object to `a` (or we give the name `a` to the Python object 12), we have the same identity:

```
a = 12
id(a)      # 140390267284112
type(a)    # <class 'int'>
```

The identity value is unique, but it will depend on the machine the program is running on.

In addition to integers, `int`, the primitive types include:

- The floating point numbers, `float`.
- 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;

A few examples:

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

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

```
id('12')            # 140388663362928
type('12')           # <class 'str'>

alphabet            # 'abcdefghijklmnopqrstuvwxyz'
id(alphabet)        # 140389733251552
type(alphabet)       # <class 'str'>
```

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
```

and their Python type:

```
type(list2)        # <class 'list'>
```

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 a complete list to a variable and a list to a list of variables as in:

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

where `list5` and `list2` refer to the same list, and `v1`, `v2`, `v3` contain, respectively, 1, 8, and 3.

## 2.6.2 List Copy

The statement `list5 = list2` creates a variable referring to the same list as `list2`. In fact, `list5` and `list2` are two names for the same object. To create a new shallow copy of a list, we need to use `list.copy()`. For example

```
list6 = list2.copy()
```

creates a new list with the same initial content as `list2`.

We can see how `copy()` works with this simple example:

```
list2          # [1, 8, 3]
list5          # [1, 8, 3] same as list2
id(list2)      # 140388791294656
id(list5)      # 140388791294656 same as list2

list6 = list2.copy()    # [1, 8, 3] initialized with
                      # the content of list2
id(list6)          # 140389740778688 new identity
```

At this point, `list2` and `list6` are only equal, while `list2` and `list5` are identical. We check the equality with `==` and the identity with `is`:

```
list2 == list5      # True
list2 == list6      # True

list2 is list5      # True
list2 is list6      # False
```

In subsequent operations affecting `list2`, `list6` will be completely independent:

```
list2[1] = 2        # [1, 2, 3]

list2          # [1, 2, 3]
list5          # [1, 2, 3] same as list2
list6          # [1, 8, 3] independent list
```

Note that `copy()` does not copy the inner objects of the list, only the identities:

```
id(list2)          # 140388791294656
id(list4.copy()[0]) # 140388791294656
```

This means that if we modify the content of `list2`, a shallow copy of `list4` will also change.

To create a deep copy of a list, or recursive copy, we need to use `copy.deepcopy()`:

```
import copy

id(copy.deepcopy(list4)[0])    # 140460465905280
```

Now the deep copy is completely independent from the original and the first item of `list4` is no longer related to `list2`.

### 2.6.3 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, 2, 3]
list3[:-1]                 # [1, 2.72, 'Perl', 'Python']
[1, 2, 3] + ['a', 'b']     # [1, 2, 3, 'a', 'b']
list2[:2] + list3[2:-1]    # [1, 2, 'Perl', 'Python']
list2 * 2                  # [1, 2, 3, 1, 2, 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, 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.

## 2.6.4 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:

```
list7 = ['a', 'b', 'c']
tuple3 = tuple(list7) # conversion to a tuple: ('a', 'b', 'c')
type(tuple3)          # <class 'tuple'>
list8 = 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, list7) # ((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']))
```

## 2.6.5 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:

```
set1 = set()                      # An empty set
set2 = {'a', 'b', 'c', 'c', 'b'}    # {'a', 'b', 'c'}
type(set2)                         # <class 'set'>
```

We can then add and remove elements with the `add()` and `remove()` functions:

```
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:

```
list9 = ['a', 'b', 'c', 'c', 'b']
set3 = set(list9)                 # {'a', 'b', 'c'}
iliad_chars = set(iliad_opening.lower())
# Set of unique characters of the iliad_opening 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']
```

## 2.6.6 Built-in Set Functions

The set methods include the classical set operations:

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

For the intersection and union operations, we can use the `&` and `|` operators instead.

A few examples:

```
set2.intersection(set3)          # {'c', 'b'}
set2 & set3                     # {'c', 'b'}
set2.union(set3)                # {'d', 'b', 'a', 'c'}
set2 | 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'}
```

### 2.6.7 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}
type(wordcount)     # <class 'dict'>
```

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
```

Another way to avoid missing key errors is to replace `dict()` with `defaultdict(default)`. `default` is often a type like `str`, `int`, `list`, or `dict`. If a `defaultdict` object accesses a key that does not exist, it creates it with the default value, i.e. the Boolean synonym of `False` for the type, respectively: `''`, `0`, `[]`, and `{}` for our examples.

```
from collections import defaultdict

missing_proof = defaultdict(int)
missing_proof['the'] # 0
```

### 2.6.8 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)])
```

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'
```

### 2.6.9 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_opening` 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_opening.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`.

Finally, differently to other languages like C, we cannot change the iteration variable inside a `for` loop in Python. For example, the loop:

```
for i in list10:
    if i == 0:
        i = 10
```

will not alter the `list10` list:

```
list10      # [0, 1, 2, 3, 4]
```

### 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. 2.6.7, 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_opening`. We also add a Boolean, `lc`, to set or not the text in lowercase:

```
def count_letters(text, lc):
    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_opening, 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_opening)` with one single parameter will be equivalent to:

```
count_letters(iliad_opening, True)
```

Functions have the type `function`:

```
type(count_letters)      # <class 'function'>
```

## 2.9 Documenting Functions

### 2.9.1 Docstrings

When writing large programs, it is essential to document the code with comments. In the case of functions, Python has a specific documentation string called a *docstring*: A string just below the function name that usually consists of a description of the function, its arguments, and what the function returns as in:

```
def count_letters(text, lc=True):
    """
    Count the letters in a text
```

```

Arguments:
    text: input text
    lc: lowercase. If true, sets the characters
        in lowercase
Returns: The letter counts
"""
...

```

Running `help(function_name)` with a function name prints its docstring:

```

>>> help(count_letters)

Count the letters in a text
...

```

## 2.9.2 Type Annotations

A second way to document a function is to give it a *signature*, i.e. annotate its arguments and return value with types as with Java or C. However, contrary to these languages, the interpreter will not check the types. They are optional and merely here to help the programmer understand and use the function more easily. We add this annotation in the first line of a function.

In the case of `count_letters()`, there are two arguments, `text` and `lc`. We annotate them with a colon followed by their types, respectively a string and a Boolean: `str` and `bool`. The function returns `letter_count`, a dictionary with a string as a key and an integer as a value. The syntax of this type is `dict[str, int]`. We insert this annotation between the list of arguments and the colon, preceded by an arrow:

```

def count_letters(text: str, lc=True: bool) -> dict[str, int]:
"""
Count the letters in a text
Arguments:
    text: input text
    lc: lowercase. If true, sets the characters
        in lowercase
Returns: The letter counts
"""
...

```

Type annotation is relatively recent in Python and still an evolving feature. We will not describe it exhaustively as the exposition can be tedious and complex. Instead, we will use it with examples when it clarifies the code.

The type annotation as well as the docstrings also apply to classes and modules that we will see later in this chapter.

## 2.10 Comprehensions and Generators

### 2.10.1 Comprehensions

Instead of loops, the comprehensions are an alternative, concise syntactic notation 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. 3.7 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: {}.

### 2.10.2 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()
```

### 2.10.3 Iterators

We just saw that we can iterate only once over a generator. Objects with this property in Python are called iterators. We can think of an iterator as an open file (a stream of data), where we read sequentially the elements until we reach the end of the file and no data is available.

We can create iterators from certain existing objects such as strings, lists, or tuples with the `iter()` function and return the next item with the `next()` function as in:

```
my_iterator = iter('abc')

next(my_iterator)      # a
next(my_iterator)      # b
next(my_iterator)      # c
```

This sequence of `next()` functions is equivalent to a `for` loop. When we reach the end and there are no more items, Python raises a `StopIteration` exception:

```
next(my_iterator)
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.

Iterators are very efficient devices as `next()` loads only one item in memory and, at the same time, probably less intuitive than lists for beginners.

## 2.10.4 zip

Let us examine now 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 = 'αβγδεζηθικλμυξοπρστυφχψω'
len(greek_alphabet)      # 24
cyrillic_alphabet = 'абвгдеёжзийклмнопрстуфхцчишъыъюя'
len(cyrillic_alphabet)    # 33
```

`zip()` weaves strings, lists, or tuples and creates an iterator of tuples, where each tuple contains the items 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[: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', 'ч')
```

Once created, we access the items with `next()` as we saw in our first example:

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

until we reach the end:

```
...
next(la_gr)    #StopIteration
```

To traverse an iterator multiple times, either we recreate it each time or we convert it to a list as in:

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

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

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[:3])
# ('a', 'α', 'а'), ('b', 'β', 'б'), ('c', 'γ', 'в')
zip(*la_gr_cy)
# ('a', 'б', 'с'), ('α', 'β', 'γ'), ('а', 'б', 'в')
```

We can use this `zip(*)` iterator to transpose efficiently a list of lists, that is to move each item of index `[i][j]` in the input to index `[j][i]` in the output. This operation is analogous to the transpose of a matrix that we will see in Sect. 5.5.8

```
la_gr_cy_list
# [('a', 'α', 'а'), ('b', 'β', 'б'), ('c', 'γ', 'в')]
list(zip(*la_gr_cy_list))
# [('a', 'б', 'с'), ('α', 'β', 'γ'), ('а', 'б', 'в')]
```

## 2.11 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
type(math)             # <class 'module'>
```

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.

## 2.12 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, *Encoding and Annotation Schemes*. 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.anaconda.com/products/individual>). Nonetheless, even if Anaconda has many packages, it may not include some modules required for certain specific programs. We will have to install them then.

## 2.13 Basic File Input/Output

Python has a set of built-in input/output functions to read and write files: `open()`, `read()`, `write()`, and `close()`. Before you run the code below, you will need a file. To follow the example, download Homer's *Iliad* and *Odyssey* from the department of classics at the Massachusetts Institute of Technology (MIT): <https://classics.mit.edu/> and store them on your computer.

The next lines open and read the `iliad.txt` file that contains Homer's *Iliad*:

```
try:
    f_iliad = open('iliad.mb.txt', 'r', encoding='utf-8')
    iliad_txt = f_iliad.read()
    f_iliad.close()
except:
    pass
```

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; In the code above, we used a try-except block in case the file does not exist or we cannot read it.

It is easy to forget to close a file. The `with` statement is a shorthand to close it automatically after the block. In the code below, we counts the letters in the text with `count_letter()` and we store the results in the `iliad_stats.txt` file: `open()` creates a new file using the write mode, `w`, and `write()` writes the results as a string.

```
iliad_stats = count_letters(iliad_txt)
with open('iliad_stats.txt', 'w') as f:
    f.write(str(iliad_stats))
    # we automatically close the file
```

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

## 2.14 Collecting a Corpus from the Internet

In the previous section, we used Homer's *Iliad* that we manually downloaded from the department of classics at the MIT and we stored it locally in the `iliad.txt` file. In practice, it is much easier to do it automatically with Python's `requests` module. Texts from classical antiquity are easily available. For example, in addition to Homer, the MIT maintains a corpus of more than 400 works from Greek and Latin authors translated into English. With the code below, we will collect five texts from this corpus, Homer's *Iliad* and *Odyssey* and Virgil's *Eclogue*, *Georgics*, and *Aeneid* and store them on our computer.

For this, we first create a dictionary with their internet addresses:

```
classics_url = {
    'iliad': 'http://classics.mit.edu/Homer/iliad.mb.txt',
    'odyssey': 'http://classics.mit.edu/Homer/odyssey.mb.txt',
    'eclogue': 'http://classics.mit.edu/Virgil/eclogue.mb.txt',
    'georgics': 'http://classics.mit.edu/Virgil/georgics.mb.txt',
    'aeneid': 'http://classics.mit.edu/Virgil/aeneid.mb.txt'}
```

We then download the texts with the `requests` module and `get()` method. We access the text content with the `text` variable. We store the corpus in a dictionary where the key is the title and the value is the text:

```
import requests

classics = {}
for key in classics_url:
    classics[key] = requests.get(classics_url[key]).text
```

All the texts from the MIT contain license information at the beginning and at the end that should not be part of the statistics. The text itself is enclosed between two dashed lines. We extract it with a regular expression and the `re.search()` function. We will describe them extensively in Chap. 3, *Corpus Processing Tools*. For now, we just run this code:

```
import regex as re

for key in classics:
    classics[key] = re.search(r'^--+$(.+)--+$',
                             classics[key],
                             re.M | re.S).group(1)

classics['iliad'][:50]
# '\n\nBOOK II\nNow the other gods and the armed warrio'
```

We can now save the corpus on our machine. A first possibility is to write the content in text files:

```
with open('iliad.txt', 'w') as f_il, \
        open('odyssey.txt', 'w') as f_od:
    f_il.write(classics['iliad'])
    f_od.write(classics['odyssey'])
```

This will not preserve the dictionary structure however and a better way is to use the Javascript object notation (JSON), a convenient data format, close to that of Python dictionaries, that will keep the structure. Here, we output the corpus in the `classics.json` file:

```
import json

with open('classics.json', 'w') as f:
    json.dump(classics, f)
```

We can subsequently reload this file with these instructions:

```
with open('classics.json', 'r') as f:
    classics = json.loads(f.read())
```

## 2.15 Memo Functions and Decorators

### 2.15.1 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]
```

## 2.15.2 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):
    ...
```

## 2.16 Object-Oriented Programming

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.

## 2.16.1 Classes and Objects

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 content, 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 `content`, `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 `content`, `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: str = None):
        """The constructor called when an object
        is created"""

        self.content = text
        self.length = len(text)
        self.letter_counts = {}

    def count_letters(self, lc: bool = True) -> dict[str, int]:
        """Function to count the letters of a text"""

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

The type of a class is `type`:

```
type(Text)      # <class 'type'>
```

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

```
txt = Text(iliad_opening)
```

The type of an object is its class:

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

We access the instance variables using this notation:

```
txt.length      # 100
```

We create and assign new instance variables the same way:

```
txt.my_var = 'a'    # a new instance variable with value 'a'
txt.content = classics['iliad']
                    # txt.content is now the value of the
                    # dictionary
```

and we call methods with the same notation:

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

Finally, we added **docstrings** and signatures to the class and its methods. We access the docstring 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.

## 2.16.2 Subclassing

Using classes, we can build a hierarchy, where the subclasses will inherit methods from their superclass parents.

Let us create a `Word` class that we define as a subclass of `Text`. Each word has a type, called a part of speech, such as verb, noun, pronoun, adjective, etc. Let us add this part of speech as an instance variable `part_of_speech` and let us add an `annotate()` function to assign a word with its part of speech. We have the new class:

```
class Word(Text):
    def __init__(self, word: str = None):
        super().__init__(word)
        self.part_of_speech = None

    def annotate(self, part_of_speech: str):
        self.part_of_speech = part_of_speech
```

where the `super().__init__(word)` function will call the constructor of `Text`.

We can then create a new word:

```
word = Word('Muse')
```

that inherits the `Text` instance variables:

```
word.length      # 4
```

and methods

```
word.count_letters(lc=False)
# {'M': 1, 'u': 1, 's': 1, 'e': 1}
```

We can also call the `Word` specific method as:

```
word.annotate('Noun')
```

and have:

```
word.part_of_speech # Noun
```

### 2.16.3 Counting with the Counter Class

The `Text` class we have created has a built-in equivalent in Python: `Counter`. It is a subclass of `dict` with counting capabilities that we can apply to any iterable: string, tuple, etc. We create a counter by passing it the iterable as an argument:

```
from collections import Counter

char_cnts = Counter(odyssey_opening)
char_cnts
# Counter({' ': 15,
#           'e': 9,
#           's': 9,
#           ...
#           '.': 1})
```

It returns a `Counter` object, here `char_cnts`, where the keys are the characters of the string or the items of the list and the values are the counts. `Counter` has all the methods of dictionaries and a few other ones. We have notably `most_common(n)` that returns the `n` most common items and `total()` that returns the sum of the values:

```
char_cnt.most_common(3)  # [(' ', 15), ('e', 9), ('s', 9)]
char_cnt.total()        # 100
```

Accessing a missing key returns a 0, but, contrary to `defaultdict`, it does not create it:

```
char_cnts['z']      # 0
'z' in char_cnts  # False
```

## 2.17 Functional Programming

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

### 2.17.1 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_opening = """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_opening, odyssey_opening])
list(text_lengths)      # [100, 111]
```

### 2.17.2 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: str) -> int:
    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)      # [807485, 610483]
```

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]] # [807485, 610483]
```

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]] # [807485, 610483]
```

### 2.17.3 `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      # 1417968
```

### 2.17.4 `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_opening` string to obtain all its vowels:

```
'.join(filter(lambda x : x in 'aeiou', iliad_opening))
# ioeeaeoieooeuaououeiaeaeaa
```

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))) # [230624, 176061]
```

## 2.18 Further Reading

Python has become very popular and there are plenty of good books or tutorials to complement this introduction. [Python.org](https://www.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. One strength of Python is the large number of libraries or modules available for it. Anaconda is a Python distribution that includes many of them<sup>1</sup>.

Python comes with an 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<sup>2</sup>. Visual Studio Code is another development environment. IPython is an interactive computing platform, where the programmer can mix code and text in the form of notebooks.

There is an impressive numbers of teaching resources on Python that range from introductions to very detailed or domain-oriented documents. Books by Matthes

---

<sup>1</sup> <https://www.anaconda.com/download/>

<sup>2</sup> <https://www.jetbrains.com/pycharm/>

(2019), Kong et al. (2020), and Lutz (2013), an older but valuable reference, are three examples of this variety. There are also many free and high-quality online courses from universities, pedagogical organizations, companies, or individuals.

Finally, the epigraph of this chapter inspires the entire book. Its origin is disputed though. For a discussion, see the blog post by Andrew Huang<sup>3</sup>:

Tracing the Origins of “I hear and I forget. I see and I remember. I do and I understand” –  
Probably Wrongly Attributed to Confucius.

---

<sup>3</sup> <https://drandrewhuang.wordpress.com/2021/05/24/tracing-the-origins-of-i-hear-and-i-forget-i-see-and-i-remember-i-do-and-i-understand-probably-wrongly-attributed-to-confucius/>

# Chapter 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.	III.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.	
<b>Aaron</b>	exo.	III.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 decades 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 billions of words are within the reach of most programmers.

**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	<i>the</i>	<i>de</i>	<i>der</i>
	<i>of</i>	<i>le</i> (article)	<i>die</i>
	<i>to</i>	<i>la</i> (article)	<i>und</i>
	<i>in</i>	<i>et</i>	<i>in</i>
	<i>and</i>	<i>les</i>	<i>des</i>
Most frequent words in Genesis	<i>and</i>	<i>et</i>	<i>und</i>
	<i>the</i>	<i>de</i>	<i>die</i>
	<i>of</i>	<i>la</i>	<i>der</i>
	<i>his</i>	à	<i>da</i>
	<i>he</i>	<i>il</i>	<i>er</i>

### 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 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 documents, paragraphs, sentences, possibly words is labeled with a semantic category or a linguistic tag, for instance a sentiment for a paragraph or a part of speech for a word. 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, parsed corpora deserve a specific mention. They are collections of syntactic structures of sentences. The production of a parsed

corpus generally requires a team of linguists to arrange the words in a dependency structure or to parenthesize the constituents of a corpus. Annotated corpora require a fair amount of handwork and are therefore more expensive than raw texts. Parsed corpora involve even more clerical work and are usually much smaller. The Universal Dependencies (de Marneffe et al., 2021) is a widely cited multilingual example of parsed corpora.

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.

**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	1 now. This beginning of miracles did Jesus in Cana of Galilee, when they saw the miracles which he did. But Jesus said, No man can do these miracles that thou doest, except it be given him, because they saw his miracles which he did on them there.
French	Galilée, le premier des miracles que fit Jésus, et il mangea, beaucoup voyant les miracles qu'il faisait, crurent qu'il ne saurait faire les miracles que vous faites, si Dieu n'est pas avec vous. Ce fut le second miracle que fit Jésus en revenant de Nazareth, parce qu'elle voyait les miracles qu'il opérait sur ceux qui étaient malades.
German	alten. Das ist das erste Zeichen, das Jesus tat, geschehe es, als du uns für ein Zeichen, daß du dies tun darfst? seinen Namen, da sie die Zeichen sahen, die er tat. Aber niemand kann die Zeichen tun, die du tust, es sei denn zu ihm: Wenn ihr nicht Zeichen und Wunder seht, so glaubt nicht an mich.

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

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

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

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

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 then a necessary step to measure performance, define specifications, and build models. Let us take a few examples:

**Measure performance.** Annotated corpora are essential tools to measure the performance of a system. The hand annotation serves as a reference when comparing it with the results of an automatic analysis. A programmer can then determine the accuracy, the robustness of an algorithm or a model, and see how well it scales up by applying it to a variety of texts.

**Define specifications.** 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.

**Learn parameters.** A third purpose of annotated corpora is to be the information source to create a model from scratch. Using statistical or machine-learning

techniques, annotated corpora enable us train the model parameters and identify those that influence the most its performance.

**Learn from raw corpora.** We saw that raw corpora could help us identify the most frequent usage of a word. Even if there is no annotation, they will also enable us to learn models. A common training procedure is to teach the model to guess words missing from a sentence or following a sequence of words. These language models, if derived from large enough corpora, will eventually capture the semantics of words.

As a summary, corpora, whether annotated or not, form the raw material of natural language processing. They are repositories from which models can derive language rules and encapsulate human knowledge.

## 3.2 Finite-State Automata

### 3.2.1 A Description

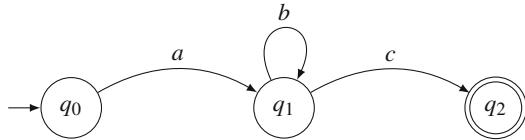
The most frequent operation we do with corpora consists in searching for words or phrases. To be convenient, search must extend beyond fixed strings. We may want to search for a word or its plural form, strings consisting of uppercase or lowercase letters, expressions containing numbers, etc. This is made possible using finite-state automata (FSA), which we introduce now. FSA are flexible tools to process texts and are one of the most adequate ways to search strings.

FSA theory was designed in the beginning of computer science as a model of abstract computing machines. It forms a well-defined formalism that has been tested and used by generations of programmers. FSA stem from a simple idea. These are devices that accept—recognize—or reject an input stream of characters. FSA are very efficient in terms of speed and memory occupation. In addition to text searching, they have many other applications: morphological parsing, part-of-speech annotation, and speech processing.

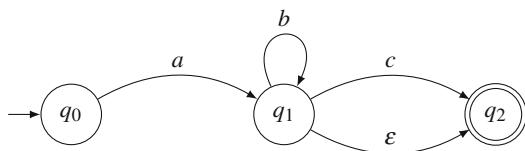
Figure 3.1 shows an automaton with three states numbered from 0 to 2, where state  $q_0$  is called the start state, and  $q_2$ , the final state. An automaton has a single start state and any number of final states, indicated by double circles. Arcs between states designate the possible transitions. Each arc is annotated by a label, which means that the transition accepts or generates the corresponding character.

An automaton accepts an input string in the following way: it starts in the initial state, follows a transition where the arc character matches the first character of the string, consumes the corresponding string character, and reaches the destination state. It then makes a second transition with the second character of the string, and continues in this way until it ends up in one of the final states and there is no character left. The automaton in Fig. 3.1 accepts or generates strings such as:  $ac$ ,  $abc$ ,  $abbc$ ,  $abbcc$ ,  $abbbbbbbbbb$ , etc. If the automaton fails to reach a final

**Fig. 3.1** A finite-state automaton



**Fig. 3.2** A finite-state automaton with an  $\varepsilon$ -transition



state, either because it has no more characters in the input string or because it is trapped in a nonfinal state, it rejects the string.

As an example, let us see how the automaton accepts string *abbc* and rejects *abbcb*. The input *abbc* is presented to the start state  $q_0$ . The first character of the string matches that of the outgoing arc. The automaton consumes character *a* and moves to state  $q_1$ . The remaining string is *bbc*. Then, the automaton loops twice on state  $q_1$  and consumes *bb*. The resulting string is character *c*. Finally, the automaton consumes *c* and reaches state  $q_2$ , which is the final state. On the contrary, the automaton does not accept string *abbcb*. It moves to states  $q_0$ ,  $q_1$ , and  $q_2$ , and consumes *abbc*. The remaining string is letter *b*. Since there is no outgoing arc with a matching symbol, the automaton is stuck in state  $q_2$  and rejects the string.

Automata may contain  $\varepsilon$ -transitions from one state to another. In this case, the automaton makes a transition without consuming any character of the input string. The automaton in Fig. 3.2 accepts strings *a*, *ab*, *abb*, etc., as well as *ac*, *abc*, *abbc*, etc.

### 3.2.2 Mathematical Definition of Finite-State Automata

FSA have a formal definition. An FSA consists of five components  $(Q, \Sigma, q_0, F, \delta)$ , where:

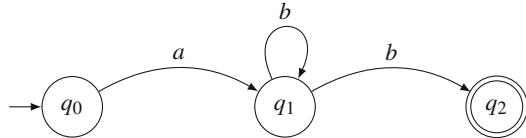
1.  $Q$  is a finite set of states.
2.  $\Sigma$  is a finite set of symbols or characters: the input alphabet.
3.  $q_0$  is the start state,  $q_0 \in Q$ .
4.  $F$  is the set of final states,  $F \subseteq Q$ .
5.  $\delta$  is the transition function  $Q \times \Sigma \rightarrow Q$ , where  $\delta(q, i)$  returns the state where the automaton moves when it is in state  $q$  and consumes the input symbol *i*.

The quintuple defining the automaton in Fig. 3.1 is  $Q = \{q_0, q_1, q_2\}$ ,  $\Sigma = \{a, b, c\}$ ,  $F = \{q_2\}$ , and  $\delta = \{\delta(q_0, a) = q_1, \delta(q_1, b) = q_1, \delta(q_1, c) = q_2\}$ . The state-transition table in Table 3.5 is an alternate representation of the  $\delta$  function.

**Table 3.5** A state-transition table where  $\emptyset$  denotes nonexistent or impossible transitions

State/Input	a	b	c
$q_0$	$q_1$	$\emptyset$	$\emptyset$
$q_1$	$\emptyset$	$q_1$	$q_2$
$q_2$	$\emptyset$	$\emptyset$	$\emptyset$

**Fig. 3.3** A nondeterministic automaton



### 3.2.3 Deterministic and Nondeterministic Automata

The automaton in Fig. 3.1 is said to be deterministic (DFA) because given a state and an input, there is one single possible destination state. On the contrary, a nondeterministic automaton (NFA) has states where it has a choice: the path is not determined in advance.

Figure 3.3 shows an example of an NFA that accepts the strings  $ab$ ,  $abb$ ,  $abbb$ , etc. Taking  $abb$  as input, the automaton reaches the state  $q_1$  consuming the letter  $a$ . Then, it has a choice between two states. The automaton can either move to state  $q_2$  or stay in state  $q_1$ . If it first moves to state  $q_2$ , there will be one character left, and the automaton will fail. The right path is to loop onto  $q_1$  and then to move to  $q_2$ .  $\epsilon$ -transitions also cause automata to be nondeterministic as in Fig. 3.2, where any string that has reached state  $q_1$  can also reach state  $q_2$ .

A possible strategy to deal with nondeterminism is to use backtracking. When an automaton has the choice between two or more states, it selects one of them and remembers the state where it made the decision: the choice point. If it subsequently fails, the automaton backtracks to the choice point and selects another state to go to. In our example in Fig. 3.3, if the automaton moves first to state  $q_2$  with the string  $bb$ , it will end up in a state without outgoing transition. It will have to backtrack and select state  $q_1$ .

### 3.2.4 Building a Deterministic Automaton from a Nondeterministic One

Although surprising, it is possible to convert any nondeterministic automaton into an equivalent deterministic automaton. We outline here an informal description of the determinization algorithm. See Hopcroft et al. (2007) for a complete description of this algorithm.

**Table 3.6** The state-transition table of the nondeterministic automaton shown in Fig. 3.3

State/Input	a	b
$q_0$	$q_1$	$\emptyset$
$q_1$	$\emptyset$	$q_1, q_2$
$q_2$	$\emptyset$	$\emptyset$

**Table 3.7** The state-transition table of the determinized automaton in Fig. 3.3

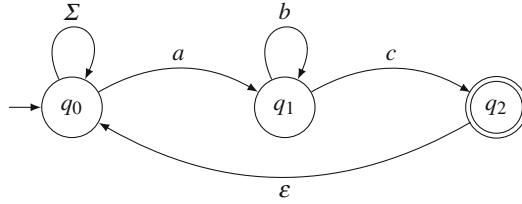
State\Input	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\{q_1, q_2\}$
$\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_0, q_2\}$	$\{q_1\}$	$\emptyset$
$\{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_1\}$	$\{q_1, q_2\}$

The algorithm starts from an NFSA  $(Q_N, \Sigma, q_0, F_N, \delta_N)$  and builds an equivalent DFSA  $(Q_D, \Sigma, \{q_0\}, F_D, \delta_D)$ , where:

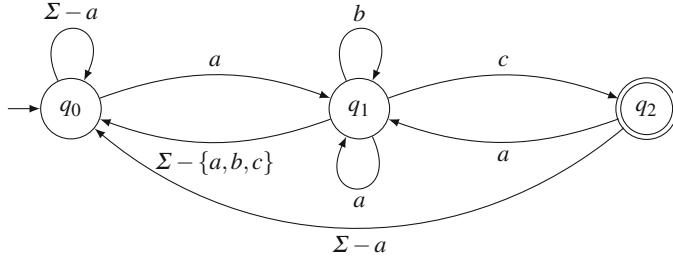
- $Q_D$  is the set of all the possible state subsets of  $Q_N$ . It is called the power set. The set of states of the automaton in Fig. 3.3 is  $Q_N = \{q_0, q_1, q_2\}$ . The corresponding set of sets is  $Q_D = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$ . If  $Q_N$  has  $n$  states,  $Q_D$  will have  $2^n$  states. In general, many of these states will be inaccessible and will be discarded.
- $F_D$  is the set of sets that include at least one final state of  $Q_D$ . In our example,  $F_D = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$ .
- For each set  $S \subset Q_N$  and for each input symbol  $a$ ,  $\delta_D(S, a) = \bigcup_{s \in S} \delta_N(s, a)$ . The state-transition table in Table 3.6 represents the automaton in Fig. 3.3. Table 3.7 represents the determinized version of it.

### 3.2.5 Searching a String with a Finite-State Automaton

Searching the occurrences of a string in a text corresponds to recognizing them with an automaton, where the string characters label the sequence of transitions. However, the automaton must skip chunks in the beginning, between the occurrences, and at the end of the text. The automaton consists then of a core accepting the searched string and of loops to process the remaining pieces. Consider again the automaton in Fig. 3.1 and modify it to search strings  $ac$ ,  $abc$ ,  $abbc$ ,  $abbac$ , etc., in a text. We add two loops: one in the beginning and the other to come back and start the search again (Fig. 3.4).



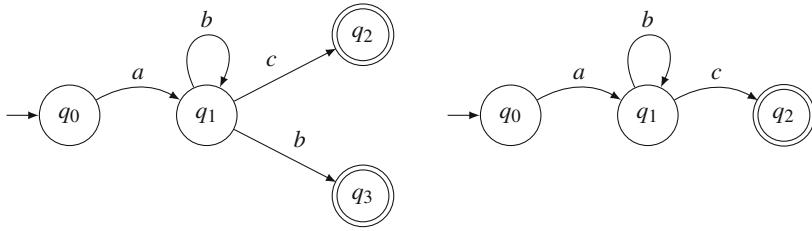
**Fig. 3.4** Searching strings  $ac$ ,  $abc$ ,  $abbc$ ,  $abbbc$ , etc.



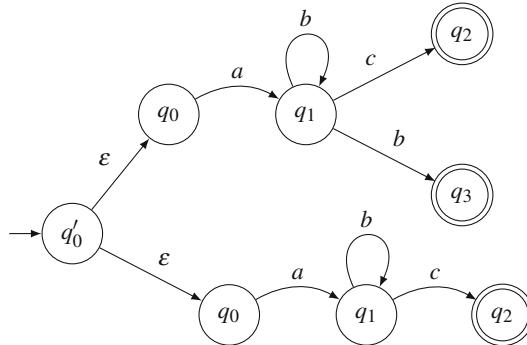
**Fig. 3.5** An automaton to search strings  $ac$ ,  $abc$ ,  $abbc$ ,  $abbbc$ , etc., in a text

In doing this, we have built an NFSA that it is preferable to convert into a DFSA. Hopcroft et al. (2007) describe the mathematical properties of such automata and an algorithm to automatically build an automaton for a given set of patterns to search. They notably report that resulting DFSA have exactly the same number of states as the corresponding NFSA. We present an informal solution to determine the transitions of the automaton in Fig. 3.4.

If the input text does not begin with an  $a$ , the automaton must consume the beginning characters and loop on the start state until it finds one. Figure 3.5 expresses this with an outgoing transition from state 0 to state 1 labeled with an  $a$  and a loop for the rest of the characters.  $\Sigma - a$  denotes the finite set of symbols except  $a$ . From state 1, the automaton proceeds if the text continues with either a  $b$  or a  $c$ . If it is an  $a$ , the preceding  $a$  is not the beginning of the string, but there is still a chance because it can start again. This corresponds to the second loop on state 1. Otherwise, if the next character falls in the set  $\Sigma - \{a, b, c\}$ , the automaton goes back to state 0. The automaton successfully recognizes the string if it reaches state 2. Then it goes back to state 0 and starts the search again, except if the next character is an  $a$ , for which it can go directly to state 1.



**Fig. 3.6** Automata A (left) and B (right)



**Fig. 3.7** The union of two automata:  $A \cup B$

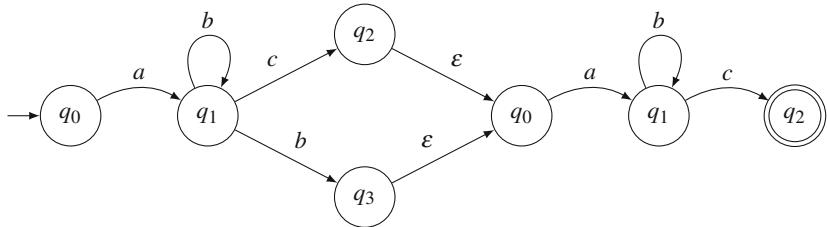
### 3.2.6 Operations on Finite-State Automata

FSA can be combined using a set of operations. The most useful are the union, the concatenation, and the closure.

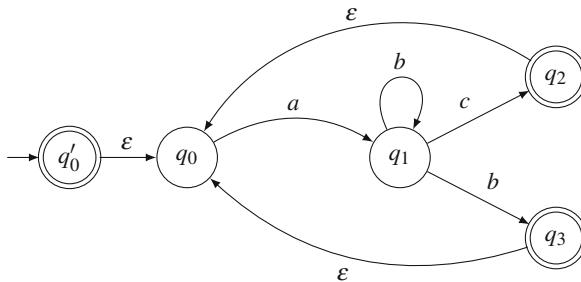
The union or sum of two automata  $A$  and  $B$  accepts or generates all the strings of  $A$  and all the strings of  $B$ . It is denoted  $A \cup B$ . We obtain it by adding a new initial state that we link to the initial states of  $A$  and  $B$  (Fig. 3.6) using  $\epsilon$ -transitions (Fig. 3.7).

The concatenation or product of  $A$  and  $B$  accepts all the strings that are concatenations of two strings, the first one being accepted by  $A$  and the second one by  $B$ . It is denoted  $A.B$ . We obtain the resulting automaton by connecting all the final states of  $A$  to the initial state of  $B$  using  $\epsilon$ -transitions (Fig. 3.8).

The iteration or Kleene closure of an automaton  $A$  accepts the concatenations of any number of its strings and the empty string. It is denoted  $A^*$ , where  $A^* = \{\epsilon\} \cup A \cup A.A \cup A.A.A \cup A.A.A.A \cup \dots$ . We obtain the resulting automaton by linking the final states of  $A$  to its initial state using  $\epsilon$ -transitions and adding a new initial state, as shown in Fig. 3.9. The new initial state enables us to obtain the empty string.



**Fig. 3.8** The concatenation of two automata:  $A.B$



**Fig. 3.9** The closure of  $A$

The notation  $\Sigma^*$  designates the infinite set of all possible strings generated from the alphabet  $\Sigma$ . Other significant operations are:

- The intersection of two automata  $A \cap B$  that accepts all the strings accepted both by  $A$  and by  $B$ . If  $A = (\Sigma, Q_1, q_1, F_1, \delta_1)$  and  $B = (\Sigma, Q_2, q_2, F_2, \delta_2)$ , the resulting automaton is obtained from the Cartesian product of states  $(\Sigma, Q_1 \times Q_2, \langle q_1, q_2 \rangle, F_1 \times F_2, \delta_3)$  with the transition function  $\delta_3(\langle s_1, s_2 \rangle, i) = \{(t_1, t_2) \mid t_1 \in \delta_1(s_1, i) \wedge t_2 \in \delta_2(s_2, i)\}$ .
- The difference of two automata  $A - B$  that accepts all the strings accepted by  $A$  but not by  $B$ .
- The complementation of the automaton  $A$  in  $\Sigma^*$  that accepts all the strings that are not accepted by  $A$ . It is denoted  $\bar{A}$ , where  $\bar{A} = \Sigma^* - A$ .
- The reversal of the automaton  $A$  that accepts all the reversed strings accepted by  $A$ .

Two automata are said to be equivalent when they accept or generate exactly the same set of strings. Useful equivalence transformations optimize computation speed or memory requirements. They include:

- $\epsilon$ -removal, which transforms an initial automaton into an equivalent one without  $\epsilon$ -transitions;
- determinization, which transforms a nondeterministic automaton into a deterministic one;

- minimization, which determines among equivalent automata the one that has the smallest number of states.

Optimization algorithms are outside the scope of this book. Hopcroft et al. (2007) as well as Roche and Schabes (1997) describe them in detail.

### 3.3 Regular Expressions

The automaton in Fig. 3.1 generates or accepts strings composed of one *a*, zero or more *b*'s, and one *c*. We can represent this set of strings using a compact notation:  $ab^*c$ , where the star symbol means any number of the preceding character. Such a notation is called a regular expression or regex. Regular expressions are very powerful devices to describe patterns to search in a text. Although their notation is different, regular expressions can always be implemented in the form of automata, and vice versa. However, regular expressions are much easier to use.

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 3.8 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
```

**Table 3.8** Examples of simple patterns and matching results

Pattern	String
<i>regular</i>	“A section on <u>regular expressions</u> ”
<i>Prolog</i>	“The <u>Prolog</u> language”
<i>the</i>	“The book of <u>the life</u> ”

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 a regex library. 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).

### 3.3.1 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 (Fig. 3.10). The star symbol is also called the closure operator or the Kleene star.

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

Fig. 3.10 Repetition metacharacters (quantifiers)

### 3.3.2 *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*, *aIe*, etc. We can combine the dot and the star in the expression `.*` to match any string of characters until we encounter a new line.

### 3.3.3 *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,m}`, and `{n,m}`. Anyway, it is always safer to use a backslash escape to avoid ambiguities.

### 3.3.4 *The Longest Match*

The description of repetition metacharacters in Fig. 3.10 sometimes makes string matching ambiguous, as with the string `aabb` 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, *Encoding and Annotation Schemes*.)

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 3.9). The regex

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

will then match the two intended strings,

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

**Table 3.9** 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

### 3.3.5 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`; `[ABCDEFGHIJKLMNPQRSTUVWXYZ]` 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 set 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-zAÄÄÆÉÈËÏÎÔÔØÙÙÝ]` 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, Perl, Java, POSIX, have predefined classes. Table 3.10 lists some you can encounter in Python programs. Some classes are adopted by all the regex variants, while some others are more specific. Their definition may vary also. For instance, `\w+` will match the accented letters in Python, but not in Perl or Java. In case of doubt, refer to the appropriate documentation.

Python's `regex` module 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 categories defined by the Unicode standard that we will review in Chap. 4, *Encoding and Annotation Schemes*. As a rule, you should always prefer the Unicode classes. They will yield the same results across the programming languages and they will enable you to handle nonLatin scripts more easily.

**Table 3.10** Some predefined character classes with their definition in Perl, after Wall et al. (2000). These classes are also available in Python's `regex` module. Note however that the content of the `\w` class is different in Python as it includes accented characters. Always prefer the Unicode classes with the `\p{...}` notation that we will see in Chap. 4, *Encoding and Annotation Schemes*

Expression	Description	Equivalent
<code>\w</code>	Any word character: letter, digit, or underscore	<code>[a-zA-Z0-9_]</code>
<code>\W</code>	Any nonword character	<code>[^ \w]</code>
<code>\s</code>	Any whitespace character: space, tabulation, new line, carriage return, or form feed	<code>[ \t\n\r\f]</code>
<code>\S</code>	Any nonwhitespace character	<code>[^\s]</code>
<code>\d</code>	Any digit	<code>[0-9]</code>
<code>\D</code>	Any nondigit	<code>[^0-9]</code>
<code>\p{L}</code>	Any Unicode letter. It includes accented letters	
<code>\P{L}</code>	Any Unicode nonletter	
<code>\p{Ll}</code>	Any Unicode lowercase letter. It includes accented letters	
<code>\p{Lu}</code>	Any Unicode uppercase letter. It includes accented letters	
<code>\p{N}</code>	Any Unicode number	
<code>\p{P}</code>	Any Unicode punctuation sign	

**Table 3.11** Some metacharacters matching nonprintable characters in Python

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

### 3.3.6 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 string using, respectively, the caret, `^`, and the dollar symbol, `$`.

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

The command line read the file line by line

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

and matches the lines of `file` 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 3.11 summarizes anchors and some nonprintable characters.

From Tables 3.11 and 2.1, you may have noted that the metacharacter `\b` was used with two different meanings: a word boundary in regular expressions or a backspace in Python strings. In fact, its interpretation depends on the context: it is a backspace in character classes; otherwise, it matches word boundaries.

### 3.3.7 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.

### 3.3.8 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*, *abcababcabc*, etc.

## 3.4 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 to incorporate them in a full-fledged programming language. We describe now, as well as the next chapter, the regular expression implementation in Python.

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

- The `m/pattern/` construct denotes a `match` operation with the regular expression `pattern`.
- The `s/pattern/replacement/` construct is a `substitution` operation. This statement matches the first occurrence of `pattern` and replaces it by the `replacement` string.
- We can add a sequence of modifiers to the `m//` and `s///` constructs just after the last `/`. For instance, if we want to replace all the occurrences of a pattern, we use the `g` modifier, where `g` stands for globally: `s/pattern/replacement/g`.

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. As Unicode is ubiquitous in natural language processing, we will use `regex` with the statement:

```
import regex as re
```

### 3.4.1 Matching

#### The `m/pattern/` Operator

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

The next program applies `m/ac*e/` to the string *The aerial acceleration alerted the ace pilot* as in Fig. 3.10:

```
import regex as re

line = 'The aerial acceleration alerted the ace pilot'
match = re.search('ac*e', line)
match      # <regex.Match object; span=(4, 6), match='ae'>
```

and finds a match object spanning from index 4 to 6 with the value `ae`.

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

```
match.group() # ae
```

## The m/pattern/g Operator

The `re.search()` function stops at the first match. If we want to implement `m/pattern/g` that finds all the matches and returns them as a list of strings, we use `findall()` instead:

```
match_list = re.findall('ac*e', line) # ['ae', 'acce', 'ace']
```

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

```
match_iter = re.finditer('ac*e', line)
list(match_iter)
# [<regex.Match object; span=(4, 6), match='ae'>,
# <regex.Match object; span=(11, 15), match='acce'>,
# <regex.Match object; span=(36, 39), match='ace'>]
```

### 3.4.2 A Simplified grep Program

In Sect. 3.3, we used the `grep` command to read files, search an expression, and print the lines where we found it. We will now write a `minigrep` program to replicate this with a `pattern`, for instance `ac*e`, given as an argument. From a Unix terminal, we will run the Python program with the command:

```
python 03_minigrep.py 'ac*e' <file_name>
```

We use a loop to read the lines and we implement `m/pattern/` with the `re.search()` function:

```
import regex as re
import sys

pattern = sys.argv[1]

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

The program first extracts the pattern to match from the command line argument `sys.argv[1]`. Then the loop 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.

### 3.4.3 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/pattern/modifiers`. `re.findall()` and `re.finditer()` that find all the occurrences of a pattern have the same flags as `re.search()`. We saw that they are equivalent to the `m/pattern/g` (globally) modifier in the PCRE framework.

Useful modifiers are:

- Case insensitive: `i`. The instruction `m/pattern/i` searches `pattern` 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 `$` match the start and the end of the input string. The instruction `m/pattern/m` considers the input string as multiple lines separated by new line characters, where the anchors `^` and `$` 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/pattern/s` match any character, including new lines.

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

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

The next program applies the patterns `m/^S/g` and `m/^s/g` to a text: It prints the letters `S` or `s` if they start a string.

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

re.findall('^S', iliad_opening) # m/^S/g
# ['S']
re.findall('^s', iliad_opening) # m/^s/g
# []
```

Only `S` starts the string. The second `.findall()` returns an empty list. We now add the case-insensitive modifier:

```
re.findall('^s', iliad_opening, re.I) # m/^s/ig
# ['S']
```

We match `S` again, but not `s` as it starts a line, but not the string. To get both, we add the multiline modifier:

```
re.findall('^s', iliad_opening, re.I | re.M) # m/^s/img
# ['S', 's']
```

and we now match `S` and `s`.

Instead of `.findall()`, we can use the `finditer()` iterator:

```
match_list = re.finditer('^s', iliad_opening, re.I | re.M)
list(match_list)
# [<regex.Match object; span=(0, 1), match='S'>,
# <regex.Match object; span=(40, 41), match='s'>]
```

### 3.4.4 Substitutions

Python uses the `re.sub()` function to substitute patterns. It has three arguments: `pattern`, `replacement`, and `string`, where the substitution occurs. It returns a new string, where by default, it substitutes all the `pattern` matches in `string` 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 `es+` with `EZ` in `iliad_opening`. This corresponds to the `s/es+/EZ/g` operation:

```
re.sub('es+', 'EZ', iliad_opening)
# Sing, O goddEZ, the anger of AchilleEZ
# son of Peleus, that brought count1EZ ills upon ...
```

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

```
# Replaces the first occurrence
line = re.sub('es+', 'EZ', iliad_opening, 1) # s/es+/EZ/
```

### 3.4.5 Backreferences

It is sometimes useful to keep a reference to matched patterns or parts of them. For example, using the *The aerial acceleration alerted the ace pilot* sentence again and the `ac+e` pattern, we have:

```
re.search('ac+e', line)
# <regex.Match object; span=(11, 15), match='acce'>
```

However, we do not know in advance how many `c` letters we will match with the `c+` pattern. To tell Python to remember it, we put parentheses around this pattern. This is called a capturing group: It creates a buffer to hold the pattern and we refer back to it by the sequence `\1`. We access its value in the matched object with the `group(1)` method:

```
match = re.search('a(c+)e', line)
match.group(1)      # 'cc'
```

We can create as many as 99 capturing groups in the current `regex` implementation. Python allocates a new buffer when it encounters a left parenthesis and refers back to it by the references `\1`, `\2`, `\3`, etc. The first pair of parentheses corresponds

to \1, the second pair to \2, the third to \3, etc. Once the pattern is applied, the \<digit> reference is returned by `group(<digit>)`:

```
match_object.group(1)
match_object.group(2)
match_object.group(3)
```

etc.

Let us change our initial pattern from `ac+e` to `.c+e`. To remember the value of what is matched by the dot, we parenthesize it as well as `c+` and we have:

```
match = re.search('(.)(c+)e', line)
match.group(1)      # 'a'
match.group(2)      # 'cc'
```

### 3.4.6 Backreferences in the Pattern

We can even use the backreferences in the pattern. For instance, 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 one.

The instruction `m/(.)\1\1/` matches such sequences. As in the previous section, we call the `group(1)` method of the returned match object to obtain the value in the buffer. Let us apply it to *acceleration* string (note the three *c*):

```
match = re.search(r'(.)\1\1', 'acceleration')
match.group(1)      # 'c'
```

### 3.4.7 Raw Strings

In the `r'(.)\1\1'` pattern, we added an `r` prefix meaning it is a *raw string*. This is to prevent Python to interpret '`\1`' as the octal value of a character.

Overall Python's regex syntax follows that of Perl. However, the escape character in Python, `\`, described in Sect. 2.4.4 has not the same meaning as the escape metacharacter from Sect. 3.3.3 in Perl regexes. Retrofitting Perl's syntax into Python regexes has led to a confusion that Python's documentation calls the *Backslash Plague*. To remedy it, and simplifying a bit, *raw strings* tell Python to interpret a string as a Perl regex.

Here are few examples of Python's interpretation of strings without and with the `r` prefix:

```
'\1'      # character code '\x01' (ASCII)
'\141'    # character code 'a'
r'\1'     # '\\1'
r'\141'   # '\\\\141'
```

### 3.4.8 Python Escape Sequences

In addition to the regular expression escape sequences, we can use the Python escape sequences defined in Table 2.1 to match nonprintable or numerically-encoded characters.

For instance, the regular expression using the Unicode number class

```
m/\p{N}+\t\p{N}+/-
```

matches two integers separated by a tabulation. The equivalent Python code applied to the *Frequencies: 100 200 300* string yields:

```
re.search(r'\p{N}+\t\p{N}+', 'Frequencies: 100 200 300')
# <regex.Match object; span=(13, 20), match='100\t200'>
```

### 3.4.9 Backreferences and Substitutions

We can finally use the backreferences in substitutions. 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(r'\$ *([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 backreferences 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(r'\$ *([0-9]+)\.?( [0-9]*)',
      r'\1 dollars and \2 cents', price)
# We'll buy it for 72 dollars and 40 cents
```

### 3.4.10 Match Objects

We saw in Sect. 3.4.1 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 parenthesized 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(r'\$ *([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:

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

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

### 3.4.11 Parameterable Regular Expressions

It is frequently the case that we need to create a regular expression from parameters like in the minigrep program in Sect. 3.4.2. This corresponds to a function that returns a regular expression. For minigrep, the function was straightforward as it used the pattern as is. Let us take a more complex example with a pattern that matches a string with a certain number of characters to the left and to the right:

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

where `string` and `width` are the function parameters, for instance:

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

To solve this, we can use `str.format()` (see Sect. 2.4.5). In our example, the function

```
def make_regex(string, width):
    return ('.{0,{width}}}{string}.{{0,{width}}}',
           .format(string=string, width=width))
```

replaces the variables with their values to produce

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

that matches the string *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.

If `string` contains metacharacters, for instance a dot as in *my string.*, we need to escape them as in: `my string\..` Otherwise, the dot would match any character. We can do this automatically with `re.escape(string)` as in:

```
string = 'my string.'
re.escape(string)
# 'my\\ string\\.'
```

Note that `re.escape()` also escapes the spaces.

We add this line to the function so that it escapes the metacharacters:

```
def make_regex(string, width):
    string = re.escape(string)
    return ('.{0,{width}}}{string}.{{0,{width}}}',
           .format(string=string, width=width))
```

Applying this regex to the *Odyssey* with the string *Penelope* yields:

```
pattern = make_regex('Penelope', 15)
re.search(pattern, odyssey, re.S).group()
# ' of his\nmother Penelope, who persist i'
```

## 3.5 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.

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 a Python program so that it can read them from the command line as in

```
python concordance.py corpus.txt pattern 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 a 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+/\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 to search the pattern. To find all the concordances in `text`, we use a `for` loop and the `re.finditer()` method that returns all the match objects in the form of an iterator. We use the start and end indices of the match object to extract and print the left and right contexts.

```
import re
import sys
```

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

# spaces match tabs and newlines
pattern = re.sub(' ', r'\s+', pattern)
# line breaks and blank sequences are replaced by spaces
text = re.sub(r'\s+', ' ', text)

for match in re.finditer(pattern, text):
    print(text[match.start() - width:match.end() + width])
```

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

## 3.6 Lookahead and Lookbehind

The `finditer()` function, just like `findall()`, scans the text from left to right and finds all the nonoverlapping matches. If we match a string and its left and right contexts with

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

the search is started again from the end index of the match (see Sect. 3.4.10).

This nonoverlapping match means that when the interval between two occurrences of `my string` is smaller than the width of the right context, `width`, it will skip the second one. In the excerpt below the two occurrences of *Hector* are six characters apart:

Meanwhile great Ajax kept on trying to drive a spear into *Hector*, but *Hector* was so skilful that he held his broad shoulders well under cover of his ox-hide shield, ever on the look-out for the whizzing of the arrows and the heavy thud of the spears.

Searching *Hector* with the regex above and a width of 20 characters will miss the second occurrence. The code below:

```

lookahead_text = 'Meanwhile great Ajax...'
pattern = make_regex(string, width)

for match in re.finditer(pattern, lookahead_text):
    print(match.group())

```

produces:

```
drive a spear into Hector, but Hector was so
```

but no second match.

To show the two lines, we need a special kind of match called a **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 regex we need is then:

```
'(.{0,20}Hector(?=.{0,20}))'
```

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

```
'.{0,20}Hector(?=(.{0,20}))'
```

The left part, corresponding to the matched object, is then stored in `group(0)` and the right one, the lookahead, in `group(1)`.

Similarly, if we want to extract the left context, we need a lookbehind construct with the syntax: `(?<=...)`. Finally, this results into the regex:

```
la_pattern= '(?<=(.{0,20}))Hector(?=(.{0,20}))'
```

and running the loop:

```
for match in re.finditer(la_pattern, lookahead_text):
    print(match.group(1), match.group(0), match.group(2))
```

we obtain now the two lines:

```
drive a spear into  Hector , but Hector was so
ar into Hector, but  Hector  was so skilful that
```

## 3.7 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.

### 3.7.1 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

**Table 3.12** 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 (Target)	Correction (Source)	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

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 3.12). They named the transformations from the point of view of the correction, not from the typo.

### 3.7.2 *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)<sup>1</sup> that uses list comprehensions: One comprehension per edit operation: delete, transpose, replace, and insert. The `edits1()` function first splits the input, the unknown word, at all possible points and then applies the operations to the list of splits. See Sect. 2.10 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 3.12 and more than 330 other strings such as: `aeress`, `hacress`, `acreccs`, `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 1 million words to build a vocabulary (see Sect. 9.4.2 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 not in the corpus.

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).

---

<sup>1</sup> Under MIT license, <https://opensource.org/license/mit/>.

### 3.7.3 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:

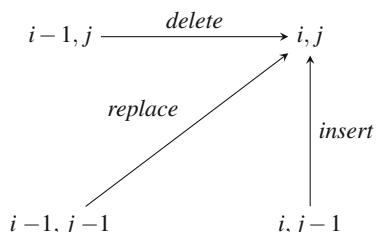
$$\text{edit\_distance}(i, j) = \min \left( \begin{array}{l} \text{edit\_distance}(i - 1, j) + \text{del\_cost} \\ \text{edit\_distance}(i - 1, j - 1) + \text{subst\_cost} \\ \text{edit\_distance}(i, j - 1) + \text{ins\_cost} \end{array} \right).$$

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

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. 3.11 represent the three edit operations, and Table 3.13 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 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.

**Fig. 3.11** Edit operations



**Table 3.13** Distances between *language* and *lineage*. The final distance is given by the value in bold at the top right of the table: 5

e	7	6	5	6	5	6	7	6	<b>5</b>
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	1	a	n	g	u	a	g	e

### 3.7.4 Computing the Minimum Edit Distance in Python

To implement the minimum edit distance in Python, we compute the length of the source and target with `len()`, we create the table as a list of lists, we initialize the first row and the first column, and we fill the table with the edit distance equation:

```
[source, target] = ('language', 'lineage')

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:
            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])
```

	First alignment	Third alignment
<b>Without epsilon symbols</b>	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
<b>With epsilon symbols</b>	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. 3.12** 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)

### 3.7.5 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**. Figure 3.12 shows two examples of them.

A frequently used technique is to consider each cell in Table 3.13 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.

### 3.8 Further Reading

Corpora are now easy to obtain. Organizations such as the Linguistic Data Consortium, ELRA, or the massive Common Crawl 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.<sup>2</sup> 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

<sup>2</sup> <https://www.corpusthomisticum.org/>

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 this 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. We will see this technique in Chap. 9, *Counting and Indexing Words*.

Text and corpus analysis are an active focus of research. Kaeding (1897) and Estoup (1912), the latter cited in Petruszewycz (1973), were among the pioneers in this field, at the turn of the twentieth 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.<sup>3</sup> Finally, the *Regular Expression 101* site<sup>4</sup> provides an excellent tool to experiment visually with regular expressions.

Although the idea of automata underlies some mathematical theories of the nineteenth 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)<sup>5</sup> library is an open-source set of functions that implements the Perl regex syntax. It is written in C by Philip Hazel. The FSA utilities<sup>6</sup> (van Noord and Gerdemann, 2001) is a Prolog package to manipulate regular expressions, automata, and transducers. The OpenFst library<sup>7</sup> (Mohri et al., 2000; Allauzen et al., 2007) is another set of tools. Both include rational operations—union, concatenation, closure, reversal—and equivalence transformation— $\varepsilon$ -elimination, determinization, and minimization.

<sup>3</sup> <https://www.regular-expressions.info/>.

<sup>4</sup> <https://regex101.com/>.

<sup>5</sup> <https://www.pcre.org/>.

<sup>6</sup> <https://www.let.rug.nl/~vannoord/Fsa/>.

<sup>7</sup> <https://www.openfst.org/>.

# Chapter 4

## Encoding and Annotation Schemes



αὐτὰρ ὁ πάσῃ  
Ἐλλάδι φωνήντα καὶ ἔμφρονα δῶρα κομίζων  
γλώσσης ὄργανα τεῦξεν ὁμόθροα, συμφυέος δὲ  
ἀρμονίης στοιχηδὸν ἐς ἄλυγα σύζυγα μίξας  
γραπτὸν ἀσιγήτοιο τύπον τορνώσατο σιγῆς,  
πάτρια θεσπεσίης δεδοημένος 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.

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, *Corpus Processing Tools* that some corpora include linguistic information to complement raw texts. This information is conveyed through annotations that describe quantities of structures. They range from the binary annotation of sentences or words to text organization, such as titles, paragraphs, and sentences, to semantic information including grammatical data 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.

In this chapter, we will examine two ways to annotate data: tables and graphs. Tables consist of rows and columns, where a row stores an observation, such as a sentence or a word, as well as its properties in different columns. The second one,

more complex, embeds the annotation in the text in the form of brackets, also called markup, eventually defining a graph structure. To create these graphs, we will use the **Extensible Markup Language** (XML), a language to define annotations, with 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, see how we can load and write tabular datasets, 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 finally create a small program to collect documents from the Internet and parse their XML structure.

## 4.1 Character Sets

### 4.1.1 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).

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 å, 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

**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	{		}	~	

**Table 4.2** Characters specific to French and German

	French	German
Lowercase	à á à ç é è ê ë ï ï ô ø û û ü ý	ä ö ü ß
Uppercase	À Á Ä Ç É È Ê Ë Ï Ï Ô Ø Û Ü Ý	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	í	ç	£	¤	¥	í	§	”	©	ª	«	¬	-	®	-	
176	º	±	²	³	’	μ	¶	.	¹	º	»	¹	²	³	¼	
192	À	Á	Â	Ã	Å	Æ	Ç	È	É	Ê	Ë	Í	Ï	Ï	Ï	
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	
224	à	á	â	ã	å	æ	ç	è	é	ê	ë	í	í	î	í	
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	

**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	í	ç	£	€	¥	Š	§	š	©	ª	«	¬	-	®	-	
176	º	±	²	³	ž	μ	¶	.	ž	¹	º	»	Œ	œ	ÿ	

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,  $\hat{e}$ , 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 macOS 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, €.

### 4.1.2 Unicode

While ASCII has been very popular, its 128 positions could not support the characters of most 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, macOS, and Python 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. The Unicode code space occupies then 24 bits out of 32 corresponding to a capacity of 1,114,112 valid 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 identifies each character or symbol by a code point and a name. The code point consists of a `U+` prefix and a single hexadecimal number, starting at `U+0000` as with:

```

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

```

We obtain the code point of a character and the character corresponding to a code point with the `ord()` and `chr()` functions, respectively:

```

ord('C'), ord('Γ')      # (67, 915)
hex(67), hex(915)       # ('0x43', '0x393')
chr(67), chr(915)       # ('C', 'Γ')

```

Unicode groups the characters of a same script in contiguous blocks of code. They start with alphabetic scripts: Latin, Greek, Cyrillic, Hebrew, Arabic, Indic, 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.

**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

### 4.1.3 Character Composition and Normalization

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  $\AA$ . 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 +  $\hat{\cdot}$  and A +  $\circ\cdot$ , 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  $\AA$ .

This may make the comparison of two characters difficult: We saw that E +  $\hat{\cdot}$  and  $\hat{E}$  have the same rendering, but how can we determine that they are equal from their Unicode encoding? Visually, both letters are identical:

```
e_1 = '\N{LATIN CAPITAL LETTER E WITH CIRCUMFLEX}' # '\u00ca'
e_2 = '\N{LATIN CAPITAL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}' # '\u0045\u0302'
```

but the code points are different:

```
e_1 == e_2 # False
```

as we have:

```
[hex(ord(cp)) for cp in e_1] # ['0xca']
[hex(ord(cp)) for cp in e_2] # ['0x45', '0x302']
```

To solve this, the Unicode standard defines normalization processes to decompose a character in a canonical sequence of code points and recompose it. This is called the normalization form decomposition (NFD) and composition (NFC). For instance,  $\hat{E}$  will be decomposed into U+0045 + U+0302 and thus equal to E and  $\circ\cdot$ . We use the

```
unicodedata.normalize(form, unistr)
```

function to carry this out. As `form`, we use ‘NFD’ to decompose the string in a canonical sequence:

```
[hex(ord(cp)) for cp in unicodedata.normalize('NFD', e_1)]
# ['0x45', '0x302']
[hex(ord(cp)) for cp in unicodedata.normalize('NFD', e_2)]
# ['0x45', '0x302']
```

and ‘NFC’ to decompose and recompose it:

```
[hex(ord(cp)) for cp in unicodedata.normalize('NFC', e_2)]
# ['0xca']
```

This will also make two looking alike characters as the angstrom unit of the length denoted Å with the U+212B code point and the Swedish Å letter equivalent. As the decomposition always follows the same fixed order, we will be able to compare strings.

```
unicodedata.normalize('NFC', e_1) == unicodedata.normalize('NFC', e_2)
# True
```

In addition to NFD and NFC, Unicode adds a compatibility concept with the normalization form compatibility decomposition (NFKD) and composition (NFKC). The *fi* ligature (U+FB01) is an example of it, where NFKD decomposes it into U+0066 + U+0069, and makes it equivalent to the sequence of two letters: *fi*. We can use the same `normalize()` function with ‘NFKC’ and ‘NFKD’ this time.

#### 4.1.4 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.

In Python, we extract the character name and category with:

```
unicodedata.name('Γ')      # 'GREEK CAPITAL LETTER GAMMA'
unicodedata.category('Γ') # 'Lu'
```

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 2.1) as:

- \N{LATIN CAPITAL LETTER E WITH CIRCUMFLEX} and \u00CA that match É 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. 3.3.5, 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 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 long or short names in Table 4.6 as respectively Letter or Lu. For example, `\p{Currency_Symbol}` matches currency symbols and `\P{L}` all nonletters.

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

Or a specific Unicode character:

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

In addition to the categories, blocks, and scripts, Unicode created properties to provide classes equivalent to the POSIX regular expressions. For example,

- `\p{space}` or `\p{Whitespace}` is equivalent to `\s`;
  - `\p{digit}` is equivalent to `\d`.

With Python's `regex` module, we can use these Unicode properties with the classes defined in Sect. 3.3.5. For instance, the class `[\p{N}\p{L}]` corresponds to the set of numbers and letters.

**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

**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_Partitian	Javanese	Kaithi	Kannada	Katakana	Kayah_Li			
Kharoshthi	Khmer	Lao	Latin	Lepcha	Limbu	Linear_B	Lisu	Lycian	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	Tifinagh	Ugaritic
Vai	Yi								

#### 4.1.5 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  $\hat{E}$  in  $F\hat{E}TE$

0046 00CA 0054 0045

or decomposing it as with  $E^+$  in  $FE^TE$

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 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  $F\hat{E}TE$  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  $\hat{E}$  is u+00CA and is in the range U-00000080 – U-000007FF. Its binary representation is 0000000011001010. UTF-8 uses the 11 rightmost bits of 00CA. The first five underlined bits together with the prefix 110

**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

form the octet **1100 0011** that corresponds to **c3** in hexadecimal. The seven next boldface bits with the prefix 10 form the octet **1000 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.

## 4.2 Locales and Word Order

### 4.2.1 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.

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.

**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

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 Å, Ä, and Ö 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, Å, Ö, Ü, and ß are considered respectively as AE, OE, UE, and ss.

### 4.2.2 The Unicode Collation Algorithm

The Unicode consortium has defined a collation algorithm (Whistler and Scherer 2023) 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, č, Č, ê, Ě, etc.} < {e, E, é, É, è, È, ê, Ĕ, 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 (́), and the umlaut (᷑). So, instances of letter E with accents, in lower- and uppercase have the order: {e, E} << {é, É} << {è, È} << {ê, Ĕ} << {ë, ĕ}, 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.

**Table 4.11** Lexical order of words with accents. Note the reversed order of the second level comparison in French

English	French
Péché	<i>pèche</i>
PÉCHÉ	<i>pêche</i>
<i>pèche</i>	<i>Pêche</i>
<i>pêche</i>	<i>Péché</i>
<i>Pêche</i>	<i>PÉCHÉ</i>
<i>pêché</i>	<i>pêché</i>
<i>Pêché</i>	<i>Pêché</i>
<i>pécher</i>	<i>pécher</i>
<i>pêcher</i>	<i>pêcher</i>

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 (Whistler and Scherer 2023) first maps the characters onto collation elements that have three numerical fields to 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.

### 4.2.3 Sorting with Python

Most operating systems have locale parameters to present information according of the linguistic or cultural rules of a user. In a Python program, we obtain this locale

values with the statements:

```
import locale

locale.getlocale()          # ('en_US', 'UTF-8')
locale.getlocale(locale.LC_CTYPE) # ('en_US', 'UTF-8')
```

here telling that the machine is using the UTF-8 encoding and it will handle characters with the US English rules. The parameters follow the POSIX standard that is organized into categories:

- `LC_COLLATE` defines how a string will be sorted;
- `LC_TIME` how to format the time;
- `LC_NUMERIC` how to format the numbers;
- `LC_MONETARY` how to format monetary values, etc.

Each category can have a specific language. Normally, we use the same language for all. We set a new locale with `setlocale()`. If we want to change all the parameter values, we use `LC_ALL` as in

```
locale.setlocale(locale.LC_ALL, 'fr_FR.UTF-8')    # 'fr_FR.UTF-8'
locale.getlocale(locale.LC_COLLATE)                 # 'fr_FR.UTF-8'
```

Note that Python just extracts these parameters from the operating system and by default these settings follow its language variable stored in `LANG`.

Using these parameters, Python can format the time, the currency, etc. in a specific language. Here we will focus on sorting strings.

By default, the Python `sorted()` function follows the code point order in the Unicode table:

```
accented = 'äëéÀËÉ'
sorted(accented)  # ['A', 'E', 'a', 'e', 'Ä', 'É', 'ä', 'é']
```

where *E* is before *a*. To sort with a locale, we set the key to `locale.strxfrm` that will transform the strings so that they can be compared with the Unicode collation algorithm. With this key, we normally have the usual lexical order:

```
sorted(accented, key=locale.strxfrm)
# ['a', 'A', 'ä', 'Ä', 'e', 'E', 'é', 'É']
```

where *a* is before *E*.

Unfortunately, the `locale` module has different implementations on different operating systems. macOS will not print the same results as Linux for instance. That is why it is preferable to use the *international components for Unicode* (ICU), which is part of the Unicode standard. ICU consists of Java and C++ classes to handle Unicode text, including collation. It has a corresponding Python module called `icu`.

The statements below create an ICU collator from the French locale and use it as a key to sort the letters. This yields the correct order on all the operating systems or programming languages:

```
import icu

collator = icu.Collator.createInstance(icu.Locale('fr_FR.UTF8'))
```

```
sorted(accented, key=collator.getSortKey)
      # ['a', 'A', 'à', 'À', 'e', 'È', 'é', 'É']
```

ICU is a very complete set of components including collation, formatting of numbers, dates, and times, conversion of encodings in Unicode, etc. Its description is beyond the scope of this book. We refer to its documentation available at <https://icu.unicode.org/> for more details.

## 4.3 Tabular Formats

Many datasets are available in the form of tabular data consisting of rows and columns. In such tables, a row represents a sample or observation and the columns contain the parameters of this sample. The construction of models in classification tasks like the analysis of sentiment, the detection of spam, or the meaning equivalence of two sentences often relies on tabular datasets.

Tabular datasets include product or movie reviews, when each row contains the text of a review and its label, for instance positive, neutral, or negative, electronic messages with the text of the message and if it is a spam or nonspam, or pairs of questions and whether they are equivalent or not. Tabular datasets may also contain the annotation of a word with its part of speech and grammatical features as we will see in Chap. 12, *Words, Parts of Speech, and Morphology*.

In this section, we will consider the Quora question pairs dataset (QQP) (Iyer et al. 2017) as it does not require any preprocessing before we can load it. QQP has more than 400,000 annotated samples and six columns, giving respectively the index of the question pair, the index of the first question, the index of the second one, the text of questions 1 and 2, and if the questions are duplicates. Table 4.12 shows examples of such pairs.

The QQP file uses tabulations to separate the values in a row. Such files have often the TSV suffix, for tabulation-separated values. The comma is another frequent separator and we have then comma-separated values or CSV files.

Python has a `csv` module that can read and write TSV and CSV files. The Pandas module can handle the same files with its `DataFrame` class and has more advanced capabilities. Let us first have a look at `csv`.

### 4.3.1 The `csv` Module

We will use the `csv.DictReader(csvfile, delimiter='\t')` class to read and format the QQP dataset, where `csvfile` is a file object or an iterator and `delimiter` is the column separator. In the QQP file, the first row contains the column names that the reader uses to format the table. If this were not the case, we would have to name the columns with a `fieldnames` argument.

**Table 4.12** An excerpt from the Quora question pair dataset. After Iyer et al. (2017)

Id	Qid1	Qid2	Question1	Question2	Is_duplicate
447	895	896	What are natural numbers?	What is a least natural number?	0
1518	3037	3038	Which pizzas are the most popularly ordered pizzas on Domino's menu?	How many calories does a Dominos pizza have?	0
3272	6542	6543	How do you start a bakery?	How one can start a bakery business?	1
3362	6722	6723	Should I learn python or Java first	If I had to choose between Java and Python, what should I choose to learn first?	1

QQP is available from the internet. We could download it and read it from a local file. Instead, we load it directly from the server as we did with the corpus of classics in Sect. 2.14. We use the `requests` module to open the connection:

```
import csv
import requests

qqp_url = 'https://qim.fs.quoracdn.net/quora_duplicate_questions.tsv'
col_names = ['id', 'qid1', 'qid2', 'question1', 'question2',
             'is_duplicate']

qqp_reader = csv.DictReader(
    requests.get(qqp_url).text.splitlines(),
    delimiter='\t')
```

We iterate over the sequence of rows and we create a list from them. The list items are dictionaries, where the keys are the column names and the values what the reader extracts from the rows:

```
>>> qqp_dataset = [row for row in qqp_reader]
>>> qqp_dataset[447]
{'id': '447',
 'qid1': '892',
 'qid2': '893',
 'question1': 'What are natural numbers?',
 'question2': 'What is a least natural number?',
 'is_duplicate': '0'}
```

The `csv` module also has a writer class, `csv.DictWriter()`, with the `writeheader()` and `writerow()` methods to write the header and the rows:

```
with open('qqp.tsv', 'w') as qqp_tsv:
    writer = csv.DictWriter(qqp_tsv, fieldnames=col_names)
    writer.writeheader()
    for row in qqp_dataset:
        writer.writerow(row)
```

### 4.3.2 Pandas

The Pandas module (McKinney 2010) is a very comprehensive suite to handle tabular data as well as time series. It can store tables in a `DataFrame` data structure, combine and query them with functions similar to those of databases. We create a reader with the statement `pandas.read_csv(filepath, sep='\t')`, where `filepath` is a file name or a file object. As with `csv`, it uses the first row to name the columns. Otherwise, we need to add a `names` argument.

We should be able to create a `DataFrame` directly from the dataset URL, unfortunately, the QQP server disallows the Pandas requests. We use then the `requests` module as for `csv`, but we have to create a memory file from the dataset string with `StringIO`:

```
import pandas as pd

qqp_pandas = pd.read_csv(
    StringIO(requests.get(qqp_url).text),
    sep='\t')
```

We extract the rows with the integer location slice method, `iloc[]`:

```
>>> qqp_pandas.iloc[447]
   id                      447
   qid1                     892
   qid2                     893
   question1           What are natural numbers?
   question2           What is a least natural number?
   is_duplicate                   0
```

and we export the dictionaries with the `to_dict('records')` method:

```
>>> qqp_pandas.iloc.to_dict('records')[447]
{'id': 447,
 'qid1': 892,
 'qid2': 893,
 'question1': 'What are natural numbers?',
 'question2': 'What is a least natural number?',
 'is_duplicate': 0}
```

We save a `DataFrame` in a file with the `to_csv` method:

```
qqp_pandas.to_csv('qqp_pd.tsv')
```

Finally, we can also save the records, the list of dictionaries making the dataset, as a JSON object as in Sect. 2.14:

```
import json

with open('qqp.json', 'w') as f:
    json.dump(qqp_pandas.to_dict('records'), f)
```

**Table 4.13** Some formatting tags in LaTeX and HTML

Language	Text in italics	New paragraph	Accented letter é
LaTeX	{\it text in italics}	\cr	\'{e}
HTML	<i>text in italics</i>	 	&acute;

## 4.4 Markup Languages

In tabular datasets, we separated the annotation from the raw text by writing it in specific columns. Sometimes, this is not very convenient as when we want to describe the property of a word or a sequence of words inside a sentence. A more intuitive practice is to incorporate the annotation of words in the text in the form of 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 Office Open XML from ISO/IEC (2016) and the (La)TeX format designed by Knuth (1986) are widely used markup languages (Table 4.13).

While TeX has been created by one person and then maintained by a group of users (TUG), Office Open XML has been adopted as an international standard. A common point to many of the current standards is that they originate in the standard markup language (SGML). 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.

### 4.4.1 An Outline of XML

XML is a markup language that defines 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 usually 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 -->
    <img></img>
    <text>Here comes the text!</text>
</book>
```

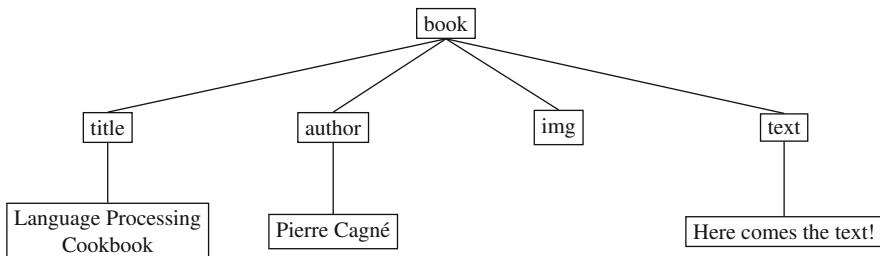
where `<book>` and `</book>` are legal tags indicating, respectively, the start and the end of the book, and `<title>` and `</title>` the beginning and the end of the title.

**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 as in Figure 4.1 for 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, when it exists, specifies the possible attributes of these elements and the value



**Fig. 4.1** Tree representation of an XML document

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>

  
```

## Entities

Finally, an entity is a name to a specific data. It can be an accented character, symbol, string as well as text or image file. The programmer declares or defines variables referring to entities in a DTD and uses them subsequently in XML documents. During parsing, the parser will replace these names by their corresponding content.

There are two types of entities: general and parameter. General entities, or simply entities, are declared in a DTD and used in XML documents. Parameter entities are only used in DTDs. The two types of entities correspond to two different contexts. They are declared and referred to differently.

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 4.14). 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.

**Table 4.14** The predefined entities of XML

Symbol	Entity encoding	Meaning
<	&lt;	Less than
>	&gt;	Greater than
&	&amp;	Ampersand
"	&quot;	Quotation mark
,	&apos;	Apostrophe

Character references consist of a Unicode hexadecimal number delimited by “`&#x`” and “`;`”, such as `&#xc9;` for É and `&#xA9;` for ©.

#### 4.4.2 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_en).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 similar to 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, macOS, and Unix, or programming languages 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 2022) is the most comprehensive document, while Whistler and Scherer (2023) describe in detail the Unicode collation algorithm. Both documents are available in electronic format from the Unicode web site<sup>1</sup>. The Unicode Consortium also maintains a public and up-to-date version of the character database<sup>2</sup>. IBM implemented a large library of Unicode components in Java and C++, which are available as open-source software<sup>3</sup>.

Pandas is a convenient tool to store, manipulate, and analyze tabular data. It has many statistical and database functionalities that makes it comparable to commercial spreadsheets. This explains why it has become popular in the Python and machine learning community. Pandas creator, McKinney (2022), wrote a book on it that is an excellent reference. There are also many Pandas tutorials on the web.

---

<sup>1</sup> <https://home.unicode.org/>

<sup>2</sup> <https://www.unicode.org/ucd/>

<sup>3</sup> <https://icu.unicode.org/>

HTML and XML markup standards are continuously evolving. Their specifications are available from the World Wide Web consortium<sup>4</sup>. HTML and XML parsers are available for most programming languages. Beautiful Soup is the most popular one for Python and has an excellent documentation<sup>5</sup>. Finally, a good reference on XML is *Learning XML* (Ray 2003).

---

<sup>4</sup> <https://www.w3.org/>

<sup>5</sup> <https://www.crummy.com/software/BeautifulSoup/>

# Chapter 5

## Python for Numerical Computations



*Calculemus!*  
Leibniz's precept

Machine learning is now essential to natural language processing. This field uses mathematical models, where vector and matrix operations are the basic tools to create or simply use algorithms. One of the compelling features of Python is its numerical module, NumPy, that facilitates considerably numerical computations. Using it, a vector sum or a matrix product is a statement that fits on one line. PyTorch is another library of linear algebra tools equipped with additional machine-learning capabilities that we will use extensively in this book.

In this chapter, we will review elementary NumPy data structures, operators, and functions, as well as those of PyTorch. We will illustrate them on a corpus of texts where we will count the characters and apply arithmetic operations and linear functions. In Sect. 2.6.7, we used dictionaries. Using this data structure, we could define functions to scale the counts or to sum them across all the texts of a corpus. We will see that we can do this much more easily with NumPy arrays or PyTorch tensors. First, we will represent the texts by vectors and the dataset with a matrix. Then, we will use the matrix to compute the pairwise similarity between the texts.

In addition to the Python modules, we will also refer to their underlying mathematical model: The vector space. I hope this will help understand the broader context of the chapter. This will not replace a serious course on linear algebra however, and, if needed, readers are invited to complement their knowledge with good tutorials on the topic.

## 5.1 Dataset

Along this chapter, we will use the corpus of classics we collected in Sect. 2.14 to exemplify the mathematical operations. To start with, we store the texts in a list of lists:

```
titles = ['iliad', 'odyssey', 'eclogue', 'georgics', 'aeneid']

texts = []
for title in titles:
    texts += [classics[title]]
```

Using the `Text` class from Sect. 2.16, we compute the character counts:

```
cnt_dicts = []
for text in texts:
    cnt_dicts += [Text(text).count_letters()]
```

and we extract the counts from the dictionaries that we store in lists:

```
cnt_lists = []
for cnt_dict in cnt_dicts:
    cnt_lists += [list(map(lambda x: cnt_dict.get(x, 0),
                           alphabet))]

cnt_lists[0][:3]      # [51016, 8938, 11558]
```

Table 5.1 shows the counts for the ten first letters.

## 5.2 Vectors

Vectors, matrices, and more generally tensors, are the fundamental mathematical objects in statistical and machine learning. We will use them to represent, store, and process datasets. Their corresponding data structure is a NumPy array or a PyTorch tensor. We start with the vectors that are tuples of  $n$  numbers, for instance 2, 3 or 26. We call these numbers the vector coordinates.

**Table 5.1** The counts of the ten first alphabet letters in the texts of the dataset

Title	a	b	c	d	e	f	g	h	i	j
Iliad	51,016	8938	11,558	28,331	77,461	16,114	12,595	50,192	38,149	1624
Odyssey	37,627	6595	8580	20,736	59,777	10,449	9803	34,785	28,793	424
Eclogue	2716	577	722	1440	4363	846	806	2508	2250	22
Georgics	6841	1618	2016	4027	12,110	2424	2147	6987	6035	59
Aeneid	36,675	6867	10,023	23,862	55,367	11,618	9606	33,055	30,576	907

### 5.2.1 Representing the Counts

Using NumPy, we create vectors of letter counts for each text of our corpus with the `array()` class. The input is a list and the output is an array object as with:

```
import numpy as np

np.array([2, 3])           # array([2, 3])
np.array([1, 2, 3])         # array([1, 2, 3])
```

For the texts in our corpus, we have:

```
iliad_cnt = np.array(cnt_lists[0])
odyssey_cnt = np.array(cnt_lists[1])
eclogue_cnt = np.array(cnt_lists[2])
georgics_cnt = np.array(cnt_lists[3])
aeneid_cnt = np.array(cnt_lists[4])

odyssey_cnt # array([37627, 6595, 8580, 20736, ...])
```

### 5.2.2 Data Types

Differently to Python lists, all the elements of a NumPy array have the same data type to optimize computations, for instance 64-bit integers, 32-bit floats, or Booleans. This type is stored in the `dtype` attribute, here a 64-bit integer:

```
odyssey_cnt.dtype # dtype('int64')
```

We optionally give it as an argument when we create the object:

```
np.array([1, 2, 3], dtype='float64')
# array([1.0, 2.0, 3.0])

np.array([0, 1, 2, 3], dtype='bool')
# array([False, True, True, True])
```

Otherwise NumPy will try to infer it from the numbers in the tuple:

```
np.array([1, 2, 3]).dtype
# dtype('int64')
```

There are many numerical datatypes in both NumPy and PyTorch and it is impossible to describe them all here. The reader should refer to the documentation.

### 5.2.3 Size of the Vectors

The `shape` attribute gives us the number of coordinates of the vector, also called the dimension of the vector space, here 26, the number of letters in the alphabet:

```
odyssey_cnt.shape # (26,)
```

### 5.2.4 Indices and Slices

We access vector coordinates in NumPy, read or write, with their indices and the slice notation just like for Python lists. Indices start at 0, following most programming languages, and contrary to the mathematical convention to start at 1.

```
vector = np.array([1, 2, 3, 4])
vector[1]    # 2
vector[:1]   # array([1])
vector[1:3]  # array([2, 3])
```

### 5.2.5 Operations

We can use the arithmetic operations, addition, subtraction, and multiplication by a scalar, on NumPy arrays:

```
np.array([1, 2, 3]) + np.array([4, 5, 6])
# array([5, 7, 9])

3 * np.array([1, 2, 3])
# array([3, 6, 9])
```

Using our dataset, we compute the character counts for Homer's works, their difference, etc.:

```
iliad_cnt + odyssey_cnt      # array([88643, 15533, 20138, ...])
iliad_cnt - odyssey_cnt      # array([13389, 2343, 2978, ...])
iliad_cnt - 2 * odyssey_cnt  # array([-24238, -4252, ...])
```

### 5.2.6 Comparison with Lists

Although the notations used in the addition and multiplication seem very intuitive, we must make sure that we are applying them to the proper data structure. Should we try to use them with Python lists, this would result in concatenations:

```
[1, 2, 3] + [4, 5, 6]
# [1, 2, 3, 4, 5, 6]

3 * [1, 2, 3]
# [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

### 5.2.7 PyTorch

PyTorch's syntax is very similar to that of NumPy. We only outline the main differences here.

#### PyTorch Tensors

The equivalent of the Numpy arrays are called tensors in PyTorch. As with NumPy, we create them from lists but this time with the `tensor()` class:

```
import torch

torch.tensor([2, 3])      # tensor([2, 3])
torch.tensor([1, 2, 3])   # tensor([1, 2, 3])
```

For a text of our dataset:

```
iliad_cnt_pt = torch.tensor(cnt_lists[0])
iliad_cnt_pt  # tensor([51016, 8938, 11558, ...])
```

#### PyTorch Data Types

The data types are also similar to those of NumPy:

```
torch.tensor([1, 2, 3]).dtype
# torch.int64

torch.tensor([1, 2, 3], dtype=torch.float64)
# tensor([1., 2., 3.], dtype=torch.float64)

torch.tensor([0, 1, 2, 3], dtype=torch.bool)
# tensor([False, True, True, True])
```

#### Size in PyTorch

PyTorch uses the method `size()` to get the number of coordinates:

```
iliad_cnt_pt.size()
# torch.Size([26])
```

#### Indices in PyTorch

The notation is the same as NumPy, except that scalars are also tensors.

```
vector = torch.tensor([1, 2, 3, 4])
vector[1]    # tensor(2)
vector[:1]   # tensor([1])
vector[1:3]  # tensor([2, 3])
```

## NumPy/PyTorch Conversion

We can convert a NumPy array to a PyTorch tensor:

```
np_array = np.array([1, 2, 3])
tensor = torch.from_numpy(np_array)
# tensor([1, 2, 3])
```

where the two variables share the same memory and vice-versa:

```
tensor = torch.tensor([1, 2, 3])
np_array = tensor.numpy()
# array([1, 2, 3])
```

## PyTorch Device

A significant difference between PyTorch and NumPy is that we can specify the processor that will carry out the computation using the `device` argument. The default device is a central processing unit (CPU), but computations are much faster with a graphics processing unit (GPU). There are multiple GPU makes that have different programming interfaces. The most popular is the Compute Unified Device Architecture (CUDA), from the Nvidia company. This is the de-facto standard in machine learning. Metal Performance Shaders (MPS) is a competitor from Apple.

We access the tensor device with the `device` attribute. By default a tensor is created on the CPU:

```
tensor = torch.tensor([1, 2, 3])
tensor.device
# tensor([1, 2, 3], device='cpu')
```

We check if a CUDA or MPS device is available on a machine with:

```
torch.cuda.is_available()
# False

torch.backends.mps.is_available()
# True
```

We set a device with this code block, where we fall back on the CPU if a GPU is not available:

```
if torch.cuda.is_available():
    device = torch.device('cuda')
elif torch.backends.mps.is_available():
    device = torch.device('mps')
else:
    device = torch.device('cpu')

device # device(type='mps')
```

We create a tensor on a specific device with:

```
tensor = torch.tensor([1, 2, 3], device=device)
```

```
tensor
# tensor([1, 2, 3], device='mps:0')
```

We move a tensor to a device with the `to()` method. With these statements, we create a tensor on the CPU and we move it on the GPU:

```
tensor = torch.tensor([1, 2, 3]) # on the CPU
tensor.to(device) # on the GPU if available
# tensor([1, 2, 3], device='mps:0')
```

### 5.2.8 Mathematical Background: The Vector Space

What we have just done can be modelled in mathematics by a vector space. In this section, we define the vocabulary and we specify a few concepts. A vector space is a set of elements called vectors. A vector space has a fixed dimension,  $n$ , and the vectors are represented as  $\mathbb{R}^n$  tuples for example:

- If  $n = 2$ ,  $\mathbb{R}^2$  vectors such as  $(2, 3)$ ,
- If  $n = 3$ ,  $\mathbb{R}^3$  vectors such as  $(1, 2, 3)$ ,
- And more generally,  $n$ -dimensional vectors in  $\mathbb{R}^n$ :  $(1, 2, 3, 4, \dots)$ ,

In our example, we used 26-dimensional vectors to hold the character counts of the texts arranged by alphabetical order.

An  $n$ -dimensional vector space has two operations called:

1. The addition of two vectors, denoted  $+$ ;
2. The multiplication of a vector by a scalar (a real number), denoted by a dot “.”.

Let  $\mathbf{u}$  and  $\mathbf{v}$  be two vectors:

$$\begin{aligned}\mathbf{u} &= (u_1, u_2, \dots, u_n), \\ \mathbf{v} &= (v_1, v_2, \dots, v_n),\end{aligned}$$

and  $\lambda$  a scalar. The results of the operations are:

1. For the addition:  $\mathbf{u} + \mathbf{v}$ , a vector:  $(u_1 + v_1, u_2 + v_2, \dots, u_n + v_n)$ ;
2. For the multiplication by a scalar:  $\lambda \cdot \mathbf{u}$ , a vector. We usually drop the dot:  $(\lambda u_1, \lambda u_2, \dots, \lambda u_n)$ .

We saw that they have straightforward implementations in NumPy and PyTorch.

## 5.3 NumPy Functions

NumPy has all the elementary mathematical functions we need in this book. When the input value is an array, the NumPy functions apply to each individual element. We say the functions are **vectorized**. Examples of mathematical functions include

the square root and the cosine. We set a precision of three digits following the decimal point:

```
np.set_printoptions(precision=3)
```

and we apply the functions:

```
np.sqrt(iliad_cnt)
       # array([225.867,  94.541, 107.508, ...])
np.cos(iliad_cnt)
       # array([-0.948, -0.986, -0.997,  0.993, ...])
```

This is to oppose to the Python functions that only apply to individual numbers (and not to lists of numbers). The statement:

```
math.sqrt(iliad_cnt) # TypeError
```

throws a type error.

We can nonetheless vectorize a Python function with `np.vectorize()`:

```
np_sqrt = np.vectorize(math.sqrt)
np_sqrt(iliad_cnt)
       # array([225.867,  94.541, 107.508, ...])
```

Python's `sum()` function returns the sum of the items in a list. NumPy will return the sum of all the elements in the array:

```
np.sum(odyssey_cnt) # 472937
```

Using the sum, we can compute the relative frequencies of the letters, first from the count vectors, either as a multiplication of a scalar by a vector,  $\lambda v$ , where  $\lambda$  is the inverse of the sum and  $v$ , the vector of counts:

```
iliad_dist = (1/np.sum(iliad_cnt)) * iliad_cnt
odyssey_dist = (1/np.sum(odyssey_cnt)) * odyssey_cnt
```

or as a division:

```
iliad_cnt / np.sum(iliad_cnt)
       # array([0.081, 0.014, 0.018, 0.045, ...])
odyssey_cnt / np.sum(odyssey_cnt)
       # array([0.08, 0.014, 0.018, 0.044, ...])
```

With the exception of `np.vectorize()`, PyTorch has similar functions that we skip in this section.

## 5.4 The Dot Product

The dot product of two vectors of equal length  $\mathbf{u} = (u_1, u_2, \dots, u_n)$  and  $\mathbf{v} = (v_1, v_2, \dots, v_n)$  is defined as  $\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$ . The corresponding NumPy function is `np.dot()` or the `@` infix operator:

```
np.dot(np.array([1, 2, 3]), np.array([4, 5, 6])) # 32
np.array([1, 2, 3]) @ np.array([4, 5, 6])      # 32
```

Let us now apply it to the two vectors of letter frequencies extracted from the *Iliad* and the *Odyssey*:

```
np.dot(iliad_dist, odyssey_dist)
# 0.06581149298284382
```

Alternatively with `@`:

```
iliad_dist @ odyssey_dist
# 0.06581149298284382
```

### 5.4.1 Euclidian Norm of a Vector

The dot product of a vector by itself defines a metric in the Euclidian vector space. Its square root is called the norm:

$$\|\mathbf{u}\| = \sqrt{\mathbf{u} \cdot \mathbf{u}}.$$

It corresponds to the magnitude of the vector (its length).

- In NumPy, we use the `np.linalg.norm()` function to compute it:

```
np.linalg.norm(np.array([1, 2, 3]))
# 3.74
```

- In PyTorch, simply `torch.linalg.vector_norm()`:

```
torch.linalg.vector_norm(torch.tensor([1.0, 2.0, 3.0]))
# tensor(3.74)
```

### 5.4.2 Cosine of Two Vectors

Finally, using the operators and functions from the previous sections, we can compute the generalized cosine of the angle between two vectors. We will apply it to numerical representations of documents or words. This is a property that we will frequently use to quantify their similarity. Given two vectors  $\mathbf{u}$  and  $\mathbf{v}$ , the cosine of their angle is defined as their dot product divided by the product of their norm:

$$\cos(\widehat{\mathbf{u}, \mathbf{v}}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|}.$$

We now apply this equation to the two vectors of letter frequencies representing the *Iliad* and the *Odyssey* and we obtain the cosine:

```
(iliad_dist @ odyssey_dist) / (
    np.linalg.norm(iliad_dist) *
    np.linalg.norm(odyssey_dist))
# 0.9990787113863588
```

A cosine of 1 corresponds to a zero angle. The 0.999 value means that the letter distributions of these two works are very close.

### 5.4.3 The Dot Product in Mathematics

In mathematics, the dot product is a bilinear form that maps two vectors to a scalar:

$$\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$$

$$f(\mathbf{u}, \mathbf{v}) = z.$$

Given two vectors  $\mathbf{u}$  and  $\mathbf{v}$ , where:

$$\begin{aligned}\mathbf{u} &= (u_1, u_2, \dots, u_n), \\ \mathbf{v} &= (v_1, v_2, \dots, v_n),\end{aligned}$$

it is defined as

$$\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i.$$

When adding such a product to a vector space, we obtain a Euclidian vector space.

### 5.4.4 Elementwise Product

We can also define an elementwise product, called the Hadamard product:

$$\mathbf{u} \odot \mathbf{v} = (u_1 v_1, u_2 v_2, \dots, u_n v_n).$$

The Hadamard product is not part of the vector space, but it is a convenient operation.

In NumPy and PyTorch, the elementwise product operator is `*`:

```
np.array([1, 2, 3]) * np.array([4, 5, 6])
# array([ 4, 10, 18])
```

It should not be confused with the dot product `@`.

## 5.5 Matrices

In statistics, data is traditionally arranged in tables, where the rows represent the observations or the objects and the columns the values of a specific attribute. In Table 5.1, each row is associated with a specific text or document and, given a row, the columns show the letter counts. This is the dataset format we follow in this book.

Skipping the first column in Table 5.1 and the first row giving the names of the texts and the letters, we have a rectangular table with a uniform numeric data type called a **matrix**. When associated with a dataset, we usually denote it  $X$ :

$$X = \begin{bmatrix} 51016 & 8938 & 11558 & 28331 & 77461 & 16114 & 12595 & 50192 & 38149 & 1624 \\ 37627 & 6595 & 8580 & 20736 & 59777 & 10449 & 9803 & 34785 & 28793 & 424 \\ 2716 & 577 & 722 & 1440 & 4363 & 846 & 806 & 2508 & 2250 & 22 \\ 6841 & 1618 & 2016 & 4027 & 12110 & 2424 & 2147 & 6987 & 6035 & 59 \\ 36675 & 6867 & 10023 & 23862 & 55367 & 11618 & 9606 & 33055 & 30576 & 907 \end{bmatrix}$$

### 5.5.1 Matrices in NumPy

Similarly to vectors, we create a matrix from the list of lists of counts, this time holding the whole dataset:

```
hv_cnts = np.array(cnt_lists)

hv_cnts # array([[51016, 8938, 11558, ...],
                  [37627, 6595, 8580, ...], ...])
```

As with vectors, the data type is inferred from the input numbers and is a 64-bit integer:

```
hv_cnts.dtype      # dtype('int64')
```

and the shape, this time, gives us the size of the matrix, here  $5 \times 26$ , as we have 5 texts represented by the counts of 26 letters:

```
hv_cnts.shape     # (5, 26)
```

### 5.5.2 Indices and Slices

As with vectors, we read or assign the elements in an array with their indices, for instance:

```
iliad_cnt[2]      # 11558
hv_cnts[1, 2]     # 8580
```

The slice operator enables us to extract parts of an array:

```
hv_cnts[:, 2]      # array([11558,  8580,   722,  2016, 10023])
hv_cnts[1, :]      # array([37627,  6595,  8580, 20736, ...])
hv_cnts[1, :2]     # array([37627,  6595])
hv_cnts[3, 2:4]    # array([2016, 4027])
hv_cnts[3:, 2:4]   # array([[ 2016,  4027],
                           [10023, 23862]])
```

A slice can also be assigned.

### 5.5.3 Order and Dimensions of a Tensor

To access an element in a vector, we need one index, for a matrix, we need two indices. More generally, we characterize multidimensional arrays, or tensors, by the number of indices. This number is called the order of the tensor, where each index refers to an axis of the array. In NumPy, the axes are numbered from 0. For a matrix, axis 0 corresponds to the vertical axis along the rows, while axis 1 is the horizontal one along the columns.

Note that there is a frequent confusion between the order, the rank, and the dimension. The order is often called (improperly) the rank. In NumPy, the number of indices of an array is called the number of dimensions and is stored in the `ndim` attribute:

```
odyssey_cnt.ndim   # 1
hv_cnts.ndim       # 2
```

A matrix is then a two-dimensional array.

In mathematics, the size of a matrix is the pair (number of rows, number of columns). They are called the dimensions of the matrix. We have seen that NumPy calls this the shape.

### 5.5.4 Addition and Multiplication by a Scalar

As with vectors, we add two matrices of identical sizes, for example matrices of size (2, 2):

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}; B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

by adding the elements with the same position:

$$A + B = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix},$$

for instance:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}.$$

We multiply a matrix by a scalar by multiplying each element:

$$\lambda A = \begin{bmatrix} \lambda a_{11} & \lambda a_{12} \\ \lambda a_{21} & \lambda a_{22} \end{bmatrix},$$

for instance:

$$0.5 \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 \\ 1.5 & 2 \end{bmatrix}.$$

Both operations have a straightforward implementation in NumPy:

```
np.array([[1, 2], [3, 4]]) + np.array([[5, 6], [7, 8]])
# array([[ 6,  8],
       [10, 12]])

0.5 * np.array([[1, 2], [3, 4]])
# array([[0.5, 1. ],
       [1.5, 2. ]])
```

Here is another example with our dataset:

```
hv_cnts - 2 * hv_cnts # array([[-51016, -8938, -11558, ...],
                               [-37627, -6595, -8580, ...], ...])
```

### 5.5.5 *Matrices in PyTorch*

In PyTorch, we create matrices with a syntax similar to that of NumPy:

```
hv_cnts_pt = torch.tensor(cnt_lists)
hv_cnts_pt # tensor([[51016, 8938, 11558, ...],
                     [37627, 6595, 8580, ...], ...])
```

We obtain the data type with:

```
hv_cnts_pt.dtype # torch.int64
```

the number of dimensions (order) with:

```
hv_cnts_pt.dim() # 2
```

the size with:

```
hv_cnts_pt.size() # torch.Size([5, 26])
```

and the size of a specific dimension with the `dim` argument name:

```
hv_cnts_pt.size(dim=0)    # 5
hv_cnts_pt.size(dim=1)    # 26
```

As with other Python indices, we can use negative numbers and -1 denotes the last dimension:

```
hv_cnts_pt.size(dim=-1)  # 26
```

### 5.5.6 Matrix Creation Functions

Both NumPy and PyTorch provide functions to create matrices of size  $(m, n)$  with specific initial values, such as

- Filled with zeros: `np.zeros((m,n))` and `torch.zeros((m,n))`;
- Filled with ones: `np.ones((m,n))` and `torch.ones((m,n))`;
- Filled with random values from a uniform distribution: `np.random.rand(m, n)` and `torch.rand(m, n)`;
- Filled with random values from a normal distribution: `np.random.randn(m, n)` and `torch.randn(m, n)`,

We create identity squared matrices with a diagonal of ones and zeros elsewhere with `np.eye(n)` and `torch.eye(n)`;

### 5.5.7 Applying Functions

As with vectors, we can apply NumPy and PyTorch functions such as the square root or the cosine to all the elements of a matrix. We can also apply them selectively to a specific dimension, for instance, to compute the sums by column to have the counts of a letter in the whole corpus or by row to have the total number of letters in a text. For this, we must specify the value of the `axis` argument in NumPy, 0 being along the rows, and hence giving the sum of each column, and 1 along the columns, and giving the sum of each row:

```
np.sum(hv_cnts)           # 1705964
np.sum(hv_cnts, axis=0)   # array([ 134875,  24595,  32899, ...])
np.sum(hv_cnts, axis=1)   # array([629980, 472937,  36313,  96739,
469995])
```

In PyTorch, we use the `dim` argument:

```
torch.sum(hv_cnts_pt)      # tensor(1705964)
torch.sum(hv_cnts_pt, dim=0) # tensor([ 134875,  24595,  32899, ...])
torch.sum(hv_cnts_pt, dim=1) # tensor([629980, 472937,  36313,  96739,
469995])
```

### 5.5.8 Transposing and Reshaping Arrays

The transpose of a matrix is an operation, where we swap the indices of the elements. The transpose of  $X = [x_{ij}]$  is denoted  $X^T = [y_{ij}]$ , where  $y_{ij} = x_{ji}$ . We transpose a matrix in NumPy or PyTorch by adding the `T` suffix:

```
hv_ctns.T # array([[51016, 37627, 2716, 6841, 36675],
                   [8938, 6595, 577, 1618, 6867],
                   [11558, 8580, 722, 2016, 10023],
                   ...])
```

Note that the transpose of a NumPy vector, a tensor of order 1, is the same vector. The transpose of `iliad_cnt`:

```
iliad_cnt.T # array([51016, 8938, 11558,...])
```

is the same as `iliad_cnt`.

Transposing a NumPy vector in a program is probably a design error. In PyTorch, this operation on tensors of dimension other than 2 is deprecated and prints a warning (it will throw an error in a future release).

Visual representations of vectors as rows or columns, such as respectively:

$$\mathbf{x} = [x_1, x_2, \dots, x_n] \text{ and } \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

are equivalent from a numeric viewpoint. If we want to make a difference, we have to create a row vector or a column vector, which are in fact matrices, and we need to state this explicitly in NumPy and in PyTorch. We create matrices of one row,  $1 \times n$  matrices, or matrices of one column,  $n \times 1$  matrices with these indices:

$$\mathbf{x} = [x_{1,1}, x_{1,2}, \dots, x_{1,n}] \text{ and } \mathbf{y} = \begin{bmatrix} y_{1,1} \\ y_{2,1} \\ \vdots \\ y_{n,1} \end{bmatrix}.$$

The elements of our row or column vectors have now two indices and we can transpose them. The transpose of a row vector is then a column vector and vice-versa.

We create a row vector by wrapping it in a list:

```
np.array([iliad_cnt]) # array([[51016, 8938, 11558, ...]])
np.array([iliad_cnt]).shape # (1, 26)
```

We then create a column vector by transposing the row vector:

```
np.array([iliad_cnt]).T # array([[51016,
                                   [8938,
                                   [11558], ...]])
```

We can also change the shape of a vector or a matrix using `reshape` with the new `shape` arguments as in:

```
iliad_cnt.reshape(1, 26)
```

to create a row vector or let simply NumPy guess a missing dimension with a `-1`:

```
iliad_cnt.reshape(1, -1) # array([[51016, 8938, 11558, ...]])
```

We create a column vector similarly:

```
iliad_cnt.reshape(-1, 1) # array([[51016], [8938], [11558], ...])
```

### 5.5.9 Reshaping with PyTorch

In PyTorch, we can use the `reshape()` method as in NumPy as well as

```
torch.unsqueeze(tensor, dim)
```

that will add one dimension at the specified `dim` index. For instance, with a `dim` value of 0, the element  $x_i$  becomes  $x_{0,i}$ :

```
torch.unsqueeze(torch.tensor([1, 2, 3]), 0)
# tensor([[1, 2, 3]])
```

And with a `dim` value of 1,  $x_i$  becomes  $x_{i,0}$ :

```
torch.unsqueeze(torch.tensor([1, 2, 3]), 1)
# tensor([[1],
[2],
[3]])
```

In addition to `unsqueeze()` and `reshape()`, PyTorch has a third function to change the size of a tensor: `view()`. `view()` has the same parameters as `reshape()` and is nearly a synonym. In a few cases, it has a different behavior though due to how PyTorch is implemented.

At the physical memory level, PyTorch stores the tensor elements in blocks and reads these elements according to the tensor size. `view()` returns a tensor with a new size telling how PyTorch must read the elements. The data will remain the same. It only changes the way it is read from the memory. To apply a `view()`, the blocks have to be contiguous, otherwise it will fail. `reshape()` always returns a tensor. It shares the data with the input tensor if the blocks are contiguous or it clones it if not. It is generally safer to use `reshape()`.

### 5.5.10 Broadcasting

So far, we have seen that we can add or subtract two matrices of equal sizes and multiply or divide a matrix by a scalar. NumPy defines additional rules to handle the addition and elementwise multiplication of matrices of different sizes. The rules automatically duplicate certain elements so that the matrices are of equal sizes and then apply the operation. This is called **broadcasting**.

In this section, we will apply broadcasting to our dataset. While we will only use a few rules, the rest follows the same principles and is quite intuitive. Some cases can be tricky however, and we refer the reader to the complete documentation if needed.

In Sect. 5.3, we used the scalar multiplication (or division) to compute the letter distribution in a text. We will replicate this for the dataset. This will involve two matrices: the counts of the letters in the texts stored in the `hv_cnts` matrix and a column vector containing the sums of the letters of each text.

Using broadcasting, the result of the multiplication of a column vector by a matrix is:

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \end{bmatrix} \odot \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & \dots \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & \dots \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & \dots \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} & \dots \\ x_{5,1} & x_{5,2} & x_{5,3} & x_{5,4} & \dots \end{bmatrix} = \begin{bmatrix} \lambda_1 x_{1,1} & \lambda_1 x_{1,2} & \lambda_1 x_{1,3} & \lambda_1 x_{1,4} & \dots \\ \lambda_2 x_{2,1} & \lambda_2 x_{2,2} & \lambda_2 x_{2,3} & \lambda_2 x_{2,4} & \dots \\ \lambda_3 x_{3,1} & \lambda_3 x_{3,2} & \lambda_3 x_{3,3} & \lambda_3 x_{3,4} & \dots \\ \lambda_4 x_{4,1} & \lambda_4 x_{4,2} & \lambda_4 x_{4,3} & \lambda_4 x_{4,4} & \dots \\ \lambda_5 x_{5,1} & \lambda_5 x_{5,2} & \lambda_5 x_{5,3} & \lambda_5 x_{5,4} & \dots \end{bmatrix},$$

where NumPy expands the vector by duplicating its elements so that it matches the matrix size. It then applies the elementwise multiplications. The arithmetic operators can be the addition, subtraction, or division.

We computed the sum of counts in the previous section, where NumPy returned a row vector. To apply the division to the rows, we need to convert these sums to a column vector:

```
np.array([np.sum(hv_cnts, axis=1)]).T # array([[629980],  
# [472937],  
# [ 36313],  
# [ 96739],  
# [469995]])
```

We can then compute the relative frequencies:

```
hv_dist = hv_cnts / np.array([np.sum(hv_cnts, axis=1)]).T  
# array([[0.081, 0.014, 0.018, ...],  
# [0.08 , 0.014, 0.018, ...],  
# [0.075, 0.016, 0.02 , ...], ...])
```

As with vectors, when we apply the elementwise product to tensors of identical sizes, we call it a Hadamard product:

$$\begin{aligned} & \left[ \begin{array}{cccccc} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & \dots \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & \dots \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & \dots \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} & \dots \\ x_{5,1} & x_{5,2} & x_{5,3} & x_{5,4} & \dots \end{array} \right] \odot \left[ \begin{array}{cccccc} y_{1,1} & y_{1,2} & y_{1,3} & y_{1,4} & \dots \\ y_{2,1} & y_{2,2} & y_{2,3} & y_{2,4} & \dots \\ y_{3,1} & y_{3,2} & y_{3,3} & y_{3,4} & \dots \\ y_{4,1} & y_{4,2} & y_{4,3} & y_{4,4} & \dots \\ y_{5,1} & y_{5,2} & y_{5,3} & y_{5,4} & \dots \end{array} \right] \\ &= \left[ \begin{array}{cccccc} x_{1,1}y_{1,1} & x_{1,2}y_{1,2} & x_{1,3}y_{1,3} & x_{1,4}y_{1,4} & \dots \\ x_{2,1}y_{2,1} & x_{2,2}y_{2,2} & x_{2,3}y_{2,3} & x_{2,4}y_{2,4} & \dots \\ x_{3,1}y_{3,1} & x_{3,2}y_{3,2} & x_{3,3}y_{3,3} & x_{3,4}y_{3,4} & \dots \\ x_{4,1}y_{4,1} & x_{4,2}y_{4,2} & x_{4,3}y_{4,3} & x_{4,4}y_{4,4} & \dots \\ x_{5,1}y_{5,1} & x_{5,2}y_{5,2} & x_{5,3}y_{5,3} & x_{5,4}y_{5,4} & \dots \end{array} \right], \end{aligned}$$

for example:

```
hv_dist * hv_dist
# array([[6.558e-03, 2.013e-04, 3.366e-04, ...],
       [6.330e-03, 1.945e-04, 3.291e-04, ...],
       [5.594e-03, 2.525e-04, 3.953e-04, ...], ...])
```

## 5.6 Matrix Products

### 5.6.1 Matrix-Vector Multiplication

The multiplication of a matrix  $X$  by a vector  $\mathbf{y}$  is a frequent operation in machine learning. This product,  $X\mathbf{y}$ , is a sequence of dot products between the matrix rows and the vector resulting in a vector:

$$X\mathbf{y} = \begin{bmatrix} X_{1,\cdot} \cdot \mathbf{y} \\ X_{2,\cdot} \cdot \mathbf{y} \\ \vdots \\ X_{n,\cdot} \cdot \mathbf{y} \end{bmatrix},$$

where  $X_{i,\cdot}$  denotes the  $i^{\text{th}}$  row of matrix  $X$ .

In its simplest form, if  $X$  consists of only one row, we multiply a row vector by a vector. The result will be a vector of one coordinate:

$$X\mathbf{y} = [x_{1,1}, x_{1,2}, \dots, x_{1,n}] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = [\sum_{i=1}^n x_{1,i} y_i].$$

With NumPy, the matrix-vector multiplication, and more generally the multiplication of matrices, uses an infix operator `@`. For the *Iliad* and the *Odyssey*, this yields:

```
hv_dist[0, :].reshape(1, -1) @ hv_dist[1, :]
# array([0.066])
```

where we reshaped the first vector to be a row vector.

Note again that the coordinate value is equal to a dot product of vectors:

```
np.dot(hv_dist[0, :], hv_dist[1, :])
```

or to

```
hv_dist[0, :] @ hv_dist[1, :]
```

## 5.6.2 Matrix Multiplication

The multiplication of two matrices  $X$  and  $Y$  of sizes  $(n, p)$  and  $(p, q)$  is a generalization of the matrix-vector multiplication to all the columns of  $Y$ . It is a matrix of size  $(n, q)$  defined as:

$$XY = \begin{bmatrix} X_{1.} \cdot Y_{.1} & X_{1.} \cdot Y_{.2} & \dots & X_{1.} \cdot Y_{.q} \\ X_{2.} \cdot Y_{.1} & X_{2.} \cdot Y_{.2} & \dots & X_{2.} \cdot Y_{.q} \\ \dots \\ X_{n.} \cdot Y_{.1} & X_{n.} \cdot Y_{.2} & \dots & X_{n.} \cdot Y_{.q} \end{bmatrix},$$

where  $Y_{.i}$  denotes the  $i^{\text{th}}$  column of matrix  $Y$ .

We compute a matrix product in NumPy and PyTorch with the `@` infix operator: `X @ Y`.

## 5.6.3 Computing the Cosines

We will now compute the cosine of all the pairs of vectors representing the works in the `hv_dist` matrix, i.e. the rows of the matrix. For this, we will first compute the dot products of all the pairs,  $\mathbf{x} \cdot \mathbf{y}$ , then the norms  $\|\mathbf{x}\|$  and  $\|\mathbf{y}\|$ , the products of the norms,  $\|\mathbf{x}\| \cdot \|\mathbf{y}\|$ , and finally the cosines,  $\frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|}$ .

1. The dot products,  $\mathbf{x} \cdot \mathbf{y}$ , of all the rows of a matrix  $X$  is simply  $XX^T$ :

```
hv_dot = hv_dist @ hv_dist.T
# array([[0.066, 0.066, 0.065, 0.066, 0.065],
       [0.066, 0.066, 0.065, 0.066, 0.065],
       [0.065, 0.065, 0.064, 0.065, 0.064],
       [0.066, 0.066, 0.065, 0.066, 0.065],
       [0.065, 0.065, 0.064, 0.065, 0.065]])
```

2. For the vector norms,  $\|\mathbf{x}\|$  and  $\|\mathbf{y}\|$ , we can use `np.linalg.norm()`. Here we will break down the computation with elementary operations. We will apply the Hadamard product,  $X \odot X$ , to have the square of the coordinates, then sum along the rows, and finally extract the square root:

```
hv_norm = np.sqrt(np.sum(hv_dist * hv_dist, axis=1))
# array([0.257, 0.257, 0.253, 0.257, 0.255])
```

3. We compute the product of the norms,  $\|\mathbf{x}\| \cdot \|\mathbf{y}\|$ , as a matrix product of a column vector by a row vector as with:

$$\begin{bmatrix} x_{1,1} \\ x_{2,1} \\ \dots \\ x_{n,1} \end{bmatrix} \begin{bmatrix} y_{1,1}, y_{1,2}, \dots, y_{1,n} \end{bmatrix} = \begin{bmatrix} x_{1,1}y_{1,1} & x_{1,1}y_{1,2} & \dots & x_{1,1}y_{1,n} \\ x_{2,1}y_{1,1} & x_{2,1}y_{1,2} & \dots & x_{2,1}y_{1,n} \\ \dots \\ x_{n,1}y_{1,1} & x_{n,1}y_{1,2} & \dots & x_{n,1}y_{1,n} \end{bmatrix}.$$

With NumPy, this corresponds to:

```
hv_norm_pairs = hv_norm.reshape(-1, 1) @ hv_norm.reshape(1, -1)
# array([[0.066, 0.066, 0.065, 0.066, 0.065],
       [0.066, 0.066, 0.065, 0.066, 0.065],
       [0.065, 0.065, 0.064, 0.065, 0.064],
       [0.066, 0.066, 0.065, 0.066, 0.065],
       [0.065, 0.065, 0.064, 0.065, 0.065]])
```

4. We are now nearly done with the cosines. We only need to divide the matrix elements by the norm products,  $\frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|}$ :

```
hv_cos = hv_dot / hv_norm_pairs
# array([[1.      , 0.999, 0.997, 0.996, 0.995],
       [0.999, 1.      , 0.997, 0.995, 0.994],
       [0.997, 0.997, 1.      , 0.996, 0.995],
       [0.996, 0.995, 0.996, 1.      , 0.998],
       [0.995, 0.994, 0.995, 0.998, 1.      ]])
```

For all the pairs, we have cosines close to 1 and thus angles close to 0. This indicates that the letter distributions are very similar. This is especially true for Homer's works.

## 5.7 Elementary Mathematical Background for Matrices

In this section, we will review quickly the mathematical background of matrices with NumPy examples.

### 5.7.1 Linear and Affine Maps

A matrix-vector multiplication corresponds to a linear map. If we add a vector to this multiplication, we have then an affine map. Both are functions between two vector spaces. Here are the simplest forms of maps:

- Linear map (function)

$$\begin{aligned}\mathbb{R} &\rightarrow \mathbb{R} \\ x &\rightarrow f(x) = ax\end{aligned}$$

- Affine map (function)

$$\begin{aligned}\mathbb{R} &\rightarrow \mathbb{R} \\ x &\rightarrow f(x) = ax + b\end{aligned}$$

Linear maps are defined by the properties:

$$\begin{aligned}f(x+y) &= f(x) + f(y) \\ f(\lambda x) &= \lambda f(x)\end{aligned}$$

### 5.7.2 Linear Functions and Vectors

We can extend linear functions to vectors, for example here with a linear combination of  $\mathbb{R}^2$  vector coordinates:

$$\begin{aligned}\mathbb{R}^2 &\rightarrow \mathbb{R}^2 \\ \mathbf{x} &\rightarrow f(\mathbf{x}) \\ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &\rightarrow \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \\ \mathbf{x} &\rightarrow A\mathbf{x} = \mathbf{y}\end{aligned}$$

With a matrix notation, we have:

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \mathbf{y}$$

This is the dot product of  $\mathbf{x}$  with each row of the matrix.

The equivalent affine map is:

$$\begin{aligned} \mathbf{x} &\rightarrow A\mathbf{x} + \mathbf{b} \\ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &\rightarrow \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \end{aligned}$$

### 5.7.3 Matrix Example

Let us give values to the linear system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= y_1 \\ a_{21}x_1 + a_{22}x_2 &= y_2 \end{aligned}$$

For instance:

$$\begin{aligned} 1 \times x_1 + 2 \times x_2 &= y_1 \\ 3 \times x_1 + 4 \times x_2 &= y_2 \end{aligned}$$

We represent it with matrices as:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

In NumPy, this corresponds to:

```
A = np.array([[1, 2],
             [3, 4]])
A @ np.array([5, 6])
# array([17, 39])
```

### 5.7.4 Transpose

We have seen that the transpose of a matrix  $A = [a_{i,j}]$  is defined as  $A^\top = [a_{j,i}]$ . This is the flipped matrix with regard to the diagonal.

We have this important property:

$$(AB)^T = B^T A^T.$$

Illustration on one example in NumPy:

```
A = np.array([[1, 2],
             [3, 4]])

B = np.array([[5, 6],
              [7, 8]])

A.T
# array([[1, 3],
        [2, 4]])

(A @ B).T
# array([[19, 43],
        [22, 50]])

B.T @ A.T
# array([[19, 43],
        [22, 50]])
```

### 5.7.5 *Matrices and Rotations*

To finish this section, let us have a look at vector rotation. From algebra courses, we know that we can use a matrix to compute a rotation of angle  $\theta$ . For a two-dimensional vector, the rotation matrix is:

$$R_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}.$$

We create a NumPy rotation matrix with an angle of  $\pi/4$ :

```
theta_45 = np.pi / 4
rot_mat_45 = np.array([[np.cos(theta_45), -np.sin(theta_45)],
                      [np.sin(theta_45), np.cos(theta_45)]])
# array([[ 0.707, -0.707],
        [ 0.707,  0.707]])
```

and we rotate vector  $(1, 1)$  by this angle with the statement:

```
rot_mat_45 @ np.array([1, 1])
# array([1.110e-16, 1.414e+00])
```

NumPy makes a roundoff error as the exact value is:

$$\begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{2} \end{bmatrix}$$

### 5.7.6 Function Composition

The composition of two functions  $f$  and  $g$  is the application of  $g$  to a variable  $x$  and then of  $f$  to  $g(x)$ . It is denoted:

$$(f \circ g)(x) = f(g(x))$$

The composition of linear maps is equal to their matrix product:

$$M_{(f \circ g)} = M_f M_g.$$

As we have seen, we compute  $M_{(f \circ g)}$  by computing the product of  $M_f$  with each column of  $M_g$ .

In NumPy and PyTorch, the operator of matrix products is `@`.

```
M_fg = M_f @ M_g
```

### 5.7.7 Application to Rotations

A sequence of rotations, for instance a rotation of  $\pi/6$  followed by a rotation of  $\pi/4$ , is a function composition. Its matrix is then simply the matrix product of the individual rotations:

$$R_{\theta_1} R_{\theta_2} = R_{\theta_1 + \theta_2},$$

here  $R_{\pi/4} R_{\pi/6} = R_{5\pi/12}$ . This product is just one line with NumPy:

```
theta_30 = np.pi / 6
rot_mat_30 = np.array([[np.cos(theta_30), -np.sin(theta_30)],
                      [np.sin(theta_30), np.cos(theta_30)]])

rot_mat_45 @ rot_mat_30
# array([[ 0.259, -0.966],
       [ 0.966,  0.259]])
```

### 5.7.8 Inverse Function

The inverse function  $g$  of  $f$  is defined as:

$$\forall x, (f \circ g)(x) = (g \circ f)(x) = x.$$

It is denoted:

$$g = f^{-1}.$$

The notation for matrices is:

$$MM^{-1} = M^{-1}M = I,$$

where  $I$  is the identity matrix with a diagonal of ones.

### 5.7.9 Inverting a Matrix in NumPy and PyTorch

In NumPy, we have the function

```
np.linalg.inv(rot_mat_30)

# array([[ 0.8660254,  0.5        ],
       [-0.5        ,  0.8660254]])
```

to invert a matrix. We check the product yields (nearly) an identity matrix:

```
np.linalg.inv(rot_mat_30) @ rot_mat_30

# array([[1.0000000e+00,  0.0000000e+00],
       [5.55111512e-17,  1.0000000e+00]])
```

In PyTorch, the function is:

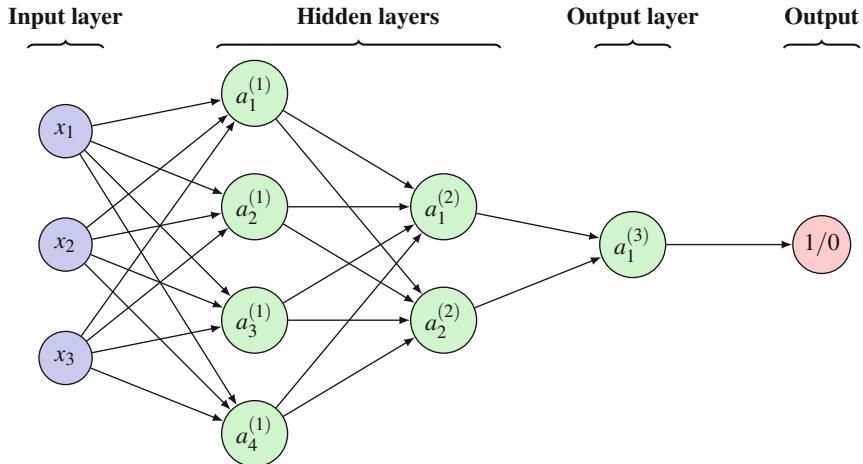
```
torch.linalg.inv(torch.from_numpy(rot_mat_30))
```

## 5.8 Application to Neural Networks

Neural networks have become very popular in natural language processing, and more generally in machine learning. While their inspiration seems to come from brain neurons, the concrete computer implementation uses matrices. In this section, we outline the link between neural networks and the content of this chapter. In Chap. 8, *Neural Networks*, we will explore more deeply the neural networks.

Figure 5.1 shows a simple neural neural network consisting of an input layer with three numeric values or nodes, then a hidden layer of four nodes, a second hidden layer of two nodes, and an output node. The information is passed through one layer to another by weighted connections. At a given layer, each node receives a weighted sum of the input nodes from the preceding layer. This is exactly a matrix-vector multiplication, where we store the weights as matrix elements.

Denoting  $\mathbf{x} = (x_1, x_2, x_3)$ , the input vector and  $\mathbf{a}^{(1)} = (a_1^{(1)}, a_2^{(1)}, a_3^{(1)}, a_4^{(1)})$ , the vector of the first hidden layer, we model the computation in the first step as an



**Fig. 5.1** A simple neural network with a binary output

affine transformation:

$$\mathbf{a}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}.$$

In neural networks, the matrix elements,  $W^{(1)}$  in the first layer, are called the weights, and the intercept, here,  $\mathbf{b}^{(1)}$ , the bias. Note that in Fig. 5.1, we did not include biases. In the rest of this section, we will set them aside to simplify the presentation.

### 5.8.1 Matrices and Datasets

The sequence of layers from left to right:

$$\mathbf{x} \rightarrow W^{(1)} \rightarrow W^{(2)} \rightarrow W^{(3)}$$

corresponds to a matrix multiplication from right to left:

$$W^{(3)} W^{(2)} W^{(1)} \mathbf{x}.$$

It is probably more intuitive to have the same order, start with  $\mathbf{x}$ , and then chain  $W^{(1)}$ ,  $W^{(2)}$ , and  $W^{(3)}$ , that is why neural networks transpose the matrices.

Representing the  $\mathbf{x}$  input as a column vector (a one-column matrix) and the first network layer by matrix  $W^{(1)}$ , we have:

$$(W^{(1)}\mathbf{x})^\top = \mathbf{x}^\top W^{(1)\top}.$$

$\mathbf{x}^\top$  is now a row vector (a one-row matrix). This corresponds to the format of a tabular dataset as we have seen in Sect. 4.3, where the observations (samples) are arranged by rows. We denote the whole dataset  $X$  and we compute the matrix product with the same order of operands:  $XW^\top$ . For the network in Fig. 5.1, we have three matrices yielding:

$$XW^{(1)\top}W^{(2)\top}W^{(3)\top} = \hat{\mathbf{y}},$$

where  $\hat{\mathbf{y}}$  denotes the output vector.

Note that the multiplication of a matrix by a vector  $W\mathbf{x}$  is a valid operation and is equal to  $\mathbf{x}W^\top$ .

### 5.8.2 Matrices and PyTorch: One Layer

We can now check that PyTorch follows this structure. We create a layer function with the `Linear` class and as arguments the size of the matrix. We set the bias to zero for sake of simplicity.

```
layer1 = torch.nn.Linear(3, 4, bias=False)
```

The weights have a random initialization and we access them with the `weight` attribute:

```
layer1.weight
# tensor([[ 0.2472, -0.4360,  0.0955],
        [-0.4775, -0.2369,  0.0147],
        [ 0.2489,  0.3770,  0.2392],
        [-0.1870, -0.0463, -0.2020]])
```

Note that these figures will differ from run to run.

We create an input vector:

```
x = torch.tensor([1.0, 2.0, 3.0])
```

and we check the output by passing  $\mathbf{x}$  to the `layer1()` function, or by using a matrix product,  $W^{(1)}\mathbf{x}$  or  $\mathbf{x}W^{(1)\top}$ . As  $\mathbf{x}$  is a vector, we do not transpose it:

```
layer1(x)
# tensor([-0.3382, -0.9072,  1.7203, -0.8855])

layer1.weight @ x
# tensor([-0.3382, -0.9072,  1.7203, -0.8855])

x @ layer1.weight.T
# tensor([-0.3382, -0.9072,  1.7203, -0.8855])
```

### 5.8.3 More Layers

In Fig. 5.1, we have three layers:

```
layer1 = torch.nn.Linear(3, 4, bias=False)
layer2 = torch.nn.Linear(4, 2, bias=False)
layer3 = torch.nn.Linear(2, 1, bias=False)
```

We check that the output from the function composition is that same as that of the matrix multiplication:

```
layer3(layer2(layer1(x)))
# tensor([0.3210])

x @ layer1.weight.T @ layer2.weight.T @ layer3.weight.T
# tensor([0.3210])
```

## 5.9 Automatic Differentiation

A difference between PyTorch and NumPy is that PyTorch tensors, i.e. scalars, vectors, or matrices here, store a record of the functions that transform them to compute automatically their gradient. Without it, we could not find the optimal network parameters that solve a problem. We will develop this in Chap. 7, *Linear and Logistic Regression*, but before that, let us examine how to extract gradients from PyTorch tensors.

As an example, let us consider a 3D curve:

$$z = x^2 + xy + y^2$$

The implementation in Python is straightforward:

```
def f(x, y):
    return x**2 + x * y + y**2
```

Its gradient is given by the partial derivatives:

$$\begin{aligned}\frac{\partial z}{\partial x} &= 2x + y, \\ \frac{\partial z}{\partial y} &= x + 2y\end{aligned}$$

and the manual computation of the gradient at (3, 4) yields:

$$\begin{aligned}\nabla f(3, 4) &= (2 \times 3 + 4, 3 + 2 \times 4), \\ &= (10, 11).\end{aligned}$$

Let us create two tensors to hold the input. We specify that we want to record the functions applied to these tensors with `requires_grad`. This will enable PyTorch to

compute the gradient:

```
x = torch.tensor(3.0, requires_grad=True)
y = torch.tensor(4.0, requires_grad=True)
```

We compute the function at this point:

```
z = f(x, y)
# tensor(37., grad_fn=<AddBackward0>)
```

PyTorch builds a graph of all the functions involved in the tensor computation. In the neural network terminology, this is called the forward pass. We extract the last function that created this variable with `grad_fn` as for `z`:

```
z.grad_fn
# <AddBackward0 at 0x1a2a66fb0>
```

From this function, we can traverse the computational graph from the end to the inputs with `next_functions`, as for the last node:

```
z.grad_fn.next_functions
# ((<AddBackward0 at 0x1a2a65780>, 0),
 # (<PowBackward0 at 0x1a2a66cb0>, 0))
```

and then recursively.

PyTorch computes the gradients with its autograd engine using the derivative chain rule. Using the recorded operations, it computes the gradient starting from the end node and backward with the `backward()` function:

```
z.backward()
```

so that we can get the gradient values with respect to the input variables:

```
x.grad, y.grad
# (tensor(10.), tensor(11.))
```

We can visualize computational graphs from tensors with the `torchviz` module and its `make_dot()` function:

```
from torchviz import make_dot

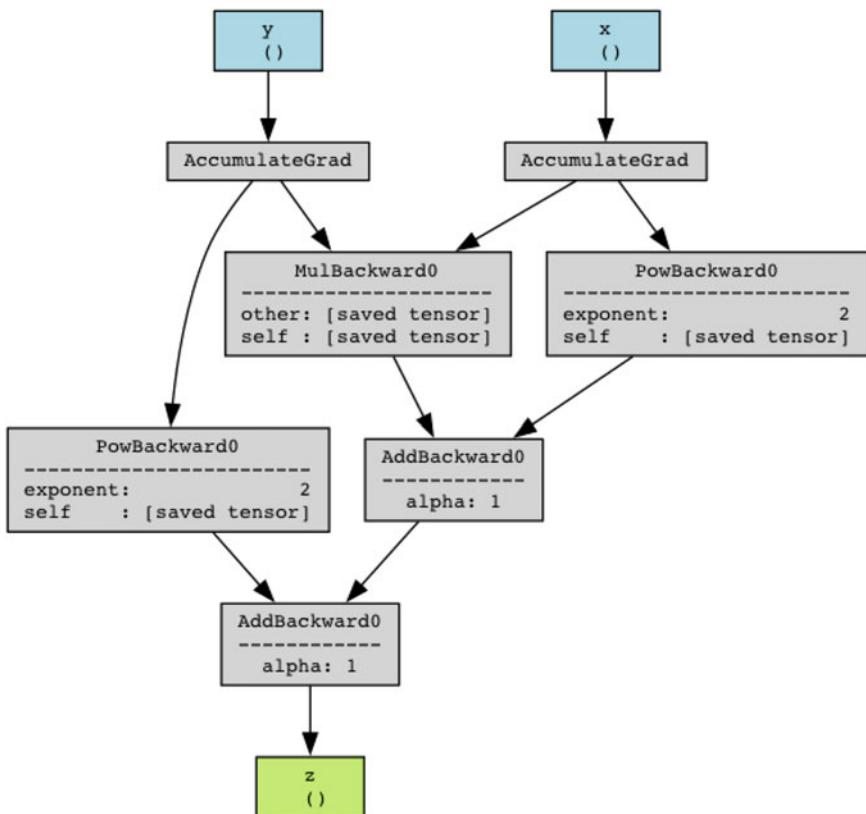
make_dot(z, params={'x': x, 'y': y, 'z': z}, show_attrs=True)
```

and Fig. 5.2 shows the graph of the  $z = f(x, y)$  function.

## 5.10 Further Reading

Linear algebra is a very large field and there are uncountable references in many languages. Any introduction will probably suffice to go deeper in this topic provided that it includes a good section on matrices.

Concerning the programming parts, there are good introductory books on NumPy, for instance the *Guide to NumPy* (Oliphant 2015) by its creator. Books on PyTorch are not as many as those on NumPy. *Deep Learning with PyTorch* (Stevens et al. 2020) is a thorough description of its features by developers who ported the initial implementation of Torch from Lua to Python.



**Fig. 5.2** PyTorch computational graph: The gray blocks in the first row from the top correspond to  $xy$  (`MulBackward`) and  $x^2$  (`PowBackward`), in the second row to  $y^2$  (`PowBackward`) and  $xy + x^2$  (`AddBackward`), and in the third one to  $y^2 + (xy + x^2)$  (`AddBackward`)

NumPy and especially PyTorch are constantly evolving. Books may be rapidly outdated and the best place to find crucial implementation details is the online documentation: [numpy.org](https://numpy.org) and [pytorch.org](https://pytorch.org). Both are excellent and up-to-date. In addition, the PyTorch tutorials are also very good.

Graphical processing units (GPU) that we introduced in Sect. 5.2.7, can accelerate considerably mathematical computations. They made it possible the development of large-scale machine–learning models. Unfortunately, they are also very expensive. In this book, we assumed that the reader only had access to a mid-range laptop and we developed all the programs so that they can run on a CPU-only machine. Readers that would like to know more on the GPU API can refer to the PyTorch documentation, notably to that of the `torch.cuda` package.<sup>1</sup>

<sup>1</sup> <https://pytorch.org/docs/stable/notes/cuda.html>.

# Chapter 6

## Topics in Information Theory and Machine Learning



Machine-learning techniques are now instrumental in most areas of natural language processing. In practical applications, we often need to determine the category of an object or an observation, such as the language in which a text is written or the part of speech of a word. We will see how we can derive mathematical models from corpora so that we can automatically detect if a paragraph is in French or in English or tell if a word is a noun or a verb.

In this chapter, we will review fundamental concepts and algorithms of machine learning. We will start with entropy, which is a ubiquitous function in information theory machine learning. Entropy will enable us to build automatically decision trees from annotated datasets or evaluate a classifier. Decision trees are simple tools to decide the category of new data or new observations, easy to understand, and, at the same time, quite efficient.

In the next chapters, we will see another family of machine-learning algorithms using linear models: logistic regression and neural networks.

### 6.1 Codes and Information Theory

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 section, 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.

### 6.1.1 Entropy

Information theory models a text as a sequence of symbols. Let  $x_1, x_2, \dots, x_N$  be a discrete set of  $N$  symbols representing the characters. The **information content** of a symbol is defined as

$$I(x_i) = -\log_2 P(x_i) = \log_2 \frac{1}{P(x_i)},$$

and it is measured in bits. When the symbols have equal probabilities, they are said to be equiprobable and

$$P(x_1) = P(x_2) = \dots = P(x_N) = \frac{1}{N}.$$

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 6.1 shows the frequency of 26 unaccented letters, the 16 accented or specific letters, and the blanks (spaces).

The entropy of the text restricted to the characters in Table 6.1 is defined as:

$$\begin{aligned} H(X) &= - \sum_{x \in X} P(x) \log_2 P(x). \\ &= -P(A) \log_2 P(A) - P(B) \log_2 P(B) - P(C) \log_2 P(C) - \dots \\ &\quad - P(Z) \log_2 P(Z) - P(\grave{A}) \log_2 P(\grave{A}) - P(\acute{A}) \log_2 P(\acute{A}) - \dots \\ &\quad - P(\ddot{U}) \log_2 P(\ddot{U}) - P(\ddot{Y}) \log_2 P(\ddot{Y}) - P(blanks) \log_2 P(blanks). \end{aligned}$$

**Table 6.1** Letter frequencies in the French novel *Salammbo* 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	5757	M	13,090	X	2206	Î	277
C	14,202	N	32,911	Y	1232	Ï	66
D	18,907	O	22,647	Z	413	Ô	397
E	71,186	P	13,161	À	1884	Œ	96
F	4993	Q	3964	Â	605	Û	179
G	5148	R	33,555	Æ	9	Û	213
H	5293	S	46,753	Ç	452	Ü	0
I	33,627	T	35,084	É	7709	Ÿ	0
J	1220	U	29,268	È	2002	Blanks	103,496
K	92	V	6916	Ê	898	Total:	593,314

If we distinguish between upper- and lowercase letters and if we include the punctuation signs, the digits, and all the other printable characters—ASCII  $\geq 32$ —the entropy of Gustave Flaubert’s *Salammbo* in French is  $H(X) = 4.370$ .

### 6.1.2 Python Implementation

The computation of entropy is easy to Python. As input, we will use a string of characters. We first normalize the text to make it easier to read and check manually the computations. For this, we replace all the spaces of the \s class with a white space. We then remove all the ASCII codes below 32. We call this function `normalize()`:

```
def normalize(corpus: str, upper: bool = False) -> str:
    corpus = re.sub(r'\s', ' ', corpus)
    corpus = re.sub(r'[\x00-\x1F]', '', corpus)
    if upper:
        corpus = corpus.upper()
    return corpus
```

Once the text is normalized, we compute the frequencies with the `Counter` class as in Sect. 2.16.3 and we divide them by the total number of characters to get the relative frequencies:

```
def rel_freqs(corpus: str) -> dict[str, float]:
    counts = Counter(corpus)
    total = counts.total()
    return {key: val/total
            for key, val in counts.items()}
```

Finally, using the relative frequencies, we compute the entropy:

```
def entropy(rel_freqs: dict[str, float]) -> float:
    entropy = 0.0
    for char in rel_freqs:
        entropy -= rel_freqs[char] * log2(rel_freqs[char])
    return entropy
```

We apply these three functions to the *Salammbo* novel in French. We store the text in the `corpus` string and we obtain:

```
>>> corpus = normalize(corpus)
>>> freqs = rel_freqs(corpus)
>>> entropy(freqs)
0.370
```

### 6.1.3 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 6.2.

The information content of equiprobable symbols is  $\log_2 8 = 3$  bits. Table 6.3 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*: -...-.

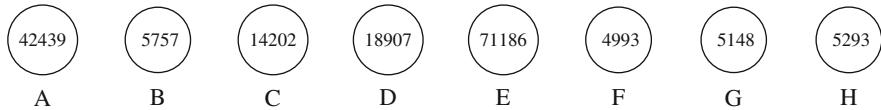
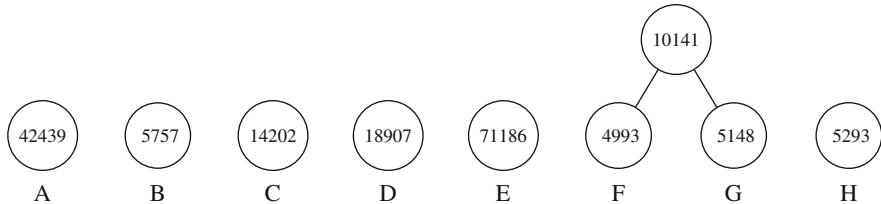
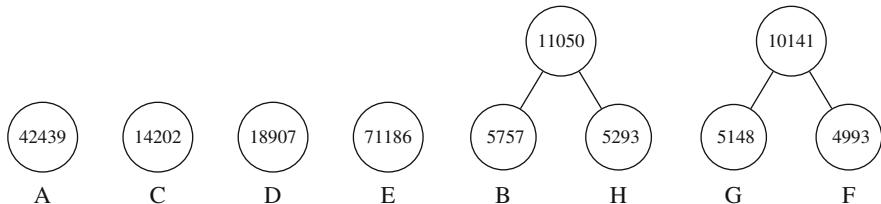
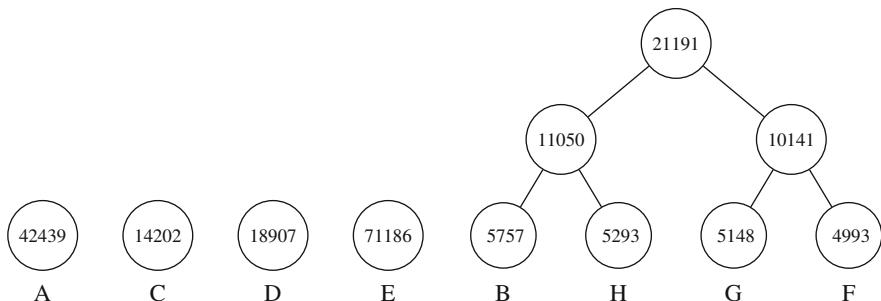
This first step builds a Huffman tree using the frequency counts. The symbols and their frequencies are the leaves of the tree. We grow the tree recursively from the leaves to the root. We merge the two symbols with the lowest frequencies into a new node that we annotate with the sum of their frequencies. In Fig. 6.1, this new node corresponds to the letters *F* and *G* with a combined frequency of  $4993 + 5148$

**Table 6.2** Frequency counts of the symbols

	A	B	C	D	E	F	G	H
Freq	42,439	5757	14,202	18,907	71,186	4993	5148	5293
Prob	0.25	0.03	0.08	0.11	0.42	0.03	0.03	0.03

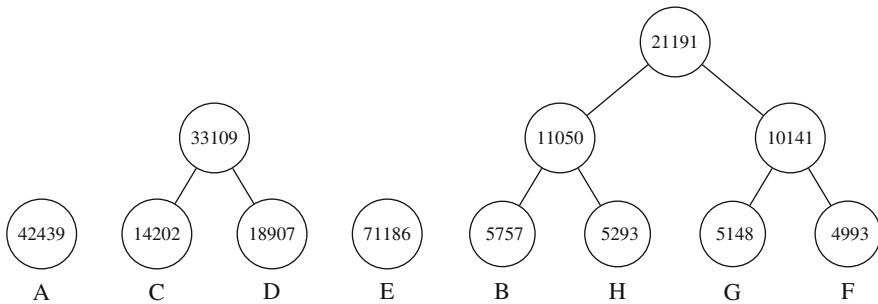
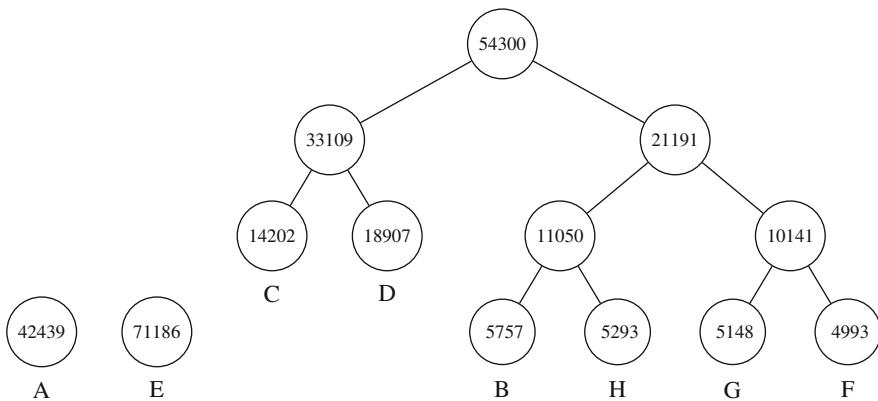
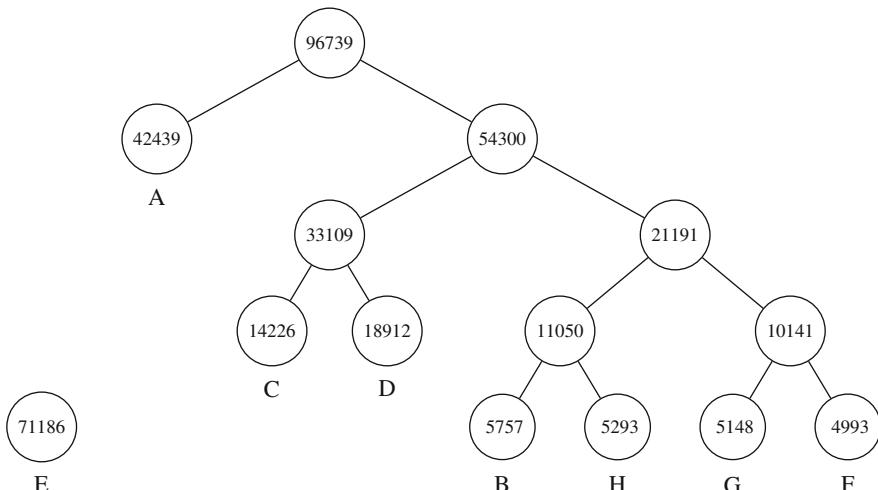
**Table 6.3** A possible encoding of the symbols on 3 bits

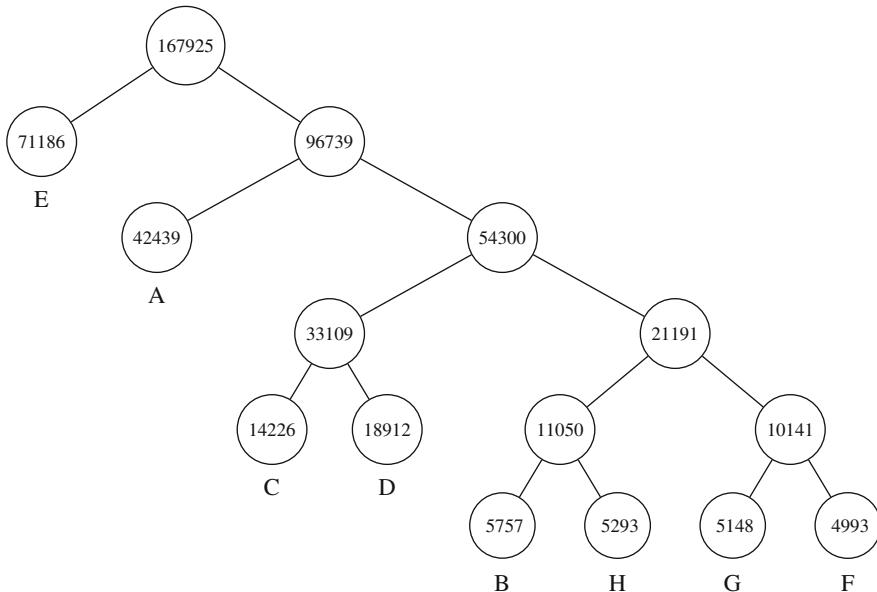
A	B	C	D	E	F	G	H
000	001	010	011	100	101	110	111

**Fig. 6.1** The symbols and their frequencies**Fig. 6.2** Merging the symbols with the lowest frequencies**Fig. 6.3** The second iteration**Fig. 6.4** The third iteration

= 10,141 (Fig. 6.2). The second iteration merges B and H (Fig. 6.3); the third one, (F, G) and (B, H) (Fig. 6.4), and so on (Figs. 6.5, 6.6, 6.7, and 6.8).

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 6.4).

**Fig. 6.5** The fourth iteration**Fig. 6.6** The fifth iteration**Fig. 6.7** The sixth iteration

**Fig. 6.8** The final Huffman tree**Table 6.4** The Huffman code

A	B	C	D	E	F	G	H
10	11100	1100	1101	0	11111	11110	11101

The average number of bits is the weighted length of a symbol. If we compute it for the data in Table 6.2, 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 6.2. It is defined by the expression:

$$-\left( \frac{42439}{167925} \log_2 \frac{42439}{167925} + \frac{5757}{167925} \log_2 \frac{5757}{167925} + \frac{14202}{167925} \log_2 \frac{14202}{167925} + \right. \\ \left. \frac{18907}{167925} \log_2 \frac{18907}{167925} + \frac{71186}{167925} \log_2 \frac{71186}{167925} + \frac{4993}{167925} \log_2 \frac{4993}{167925} + \right. \\ \left. \frac{5148}{167925} \log_2 \frac{5148}{167925} + \frac{5293}{167925} \log_2 \frac{5293}{167925} \right) = 2.31$$

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.

### 6.1.4 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 twenty-first 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  $M(x_1), M(x_2), \dots, M(x_N)$ . The other distribution,  $P$ , corresponds to the test data:  $P(x_1), P(x_2), \dots, P(x_N)$ . 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 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.

In the definition of cross-entropy, the sum is over the set  $X$  of symbols in  $M$  and  $P$ . However some of our test texts have characters that are not in the *Salammbô* training set. For instance, *Notre Dame de Paris* contains Greek letters, while *Salammbô* has only Latin characters. For such Greek letters,  $M(x) = 0$  and, as  $-\log_2 0$  is infinite, this would result in an infinite cross-entropy. In practical cases, we should avoid this situation and find a way to deal with unknown symbols. We will discuss techniques how to smooth distributions in Chap. 10, *Word Sequences*. In this experiment, we restricted  $X$  to the symbols occurring in the training set.

**Table 6.5** The entropies are measured on the test sets and the cross-entropies are measured with Chapters 1–14 of Gustave Flaubert’s *Salammbô* taken as the model

	Train/Test	Entropy	Cross-entropy	Difference
		$H(P)$	$H(P M)$	$H(P, M) - H(P)$
<i>Salammbô</i> , chapters 1–14	Training set ( $M$ )	4.37168	4.37168	0.00000
<i>Salammbô</i> , chapter 15	Test set ( $P$ )	4.31338	4.32544	0.01206
<i>Notre Dame de Paris</i>	Test set ( $P$ )	4.42285	4.44187	0.01889
<i>Nineteen eighty-four</i>	Test set ( $P$ )	4.34982	4.79617	0.44635

**Table 6.6** The perplexity and cross-perplexity of texts measured with Chapters 1–14 of Gustave Flaubert’s *Salammbô* taken as the model

	Train/Test	Perplexity	Cross-perplexity
<i>Salammbô</i> , chapters 1–14	Training set	20.70	20.70
<i>Salammbô</i> , chapter 15	Test set	19.88	20.05
<i>Notre Dame de Paris</i>	Test set	21.45	21.73
<i>Nineteen eighty-four</i>	Test set	20.39	27.78

Using this simplification, the data in Table 6.5 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*.

### 6.1.5 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 6.6 shows the perplexity and cross-perplexity of the same texts measured with Chaps. 1–14 of Gustave Flaubert’s *Salammbô* taken as the model.

## 6.2 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

**Table 6.7** A dataset of objects member of two classes:  $P$  and  $N$ . 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	$N$

resembles a reverse Huffman encoding, is one of the simplest machine-learning techniques.

### 6.2.1 A Toy Dataset

Machine learning considers collections of objects or observations, called **datasets**, where each object is defined by a set of attributes,  $SA$ . Each attribute has a set of possible values called the attribute domain. Table 6.7 from Quinlan (1986) shows a small dataset of 14 weather observations, where each observation is described by four attributes:

$$SA = \{Outlook, Temperature, Humidity, Windy\}.$$

The attributes have the following respective domains:

- $\text{dom}(Outlook) = \{\text{sunny}, \text{overcast}, \text{rain}\}$ ,
- $\text{dom}(Temperature) = \{\text{hot}, \text{mild}, \text{cool}\}$ ,
- $\text{dom}(Humidity) = \{\text{normal}, \text{high}\}$ ,
- $\text{dom}(Windy) = \{\text{true}, \text{false}\}$ .

Each observation is member of a class,  $P$  or  $N$  in this dataset, corresponding to days suitable for playing tennis ( $P$ ) or not ( $N$ ).

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

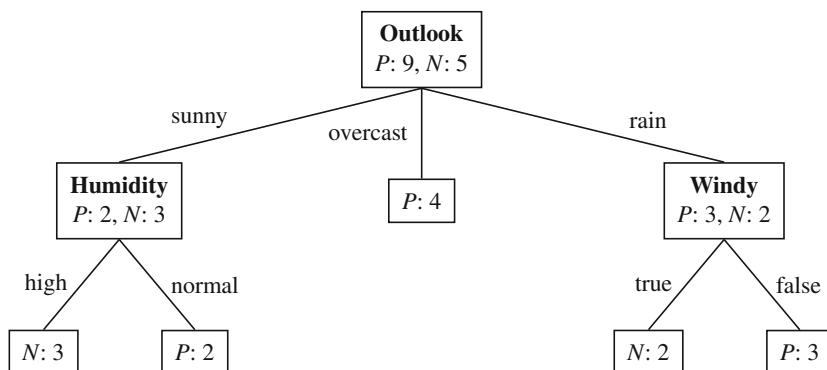
set, here the examples in Fig. 6.7. Once the tree is induced, it will be able to predict the class of examples taken outside the training set.

Machine-learning techniques make it possible to build models that 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 most NLP applications and tasks, including text classification, part-of-speech tagging, group detection, named entity recognition, or translation, that we will describe in the next chapters of this book.

### 6.2.2 Decision Trees

A decision tree is a tool to classify objects such as those in Table 6.7. 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.

Figure 6.9 shows a decision tree that correctly classifies all the objects in the set shown in Table 6.7 (Quinlan 1986).



**Fig. 6.9** A decision tree classifying the objects in Table 6.7. 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)

### 6.2.3 Inducing Decision Trees Automatically

It is possible to design many trees that classify successfully the objects in Table 6.7. The tree in Fig. 6.9 is interesting because it is efficient: a decision can be made with a minimal number of tests.

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 (Sect. 6.1.1), for a two-class set  $X = \{P, N\}$  of respectively  $p$  positive and  $n$  negative examples, it is:

$$H(X) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}.$$

Figure 6.10 shows this binary entropy function with  $x = \frac{p}{p+n}$ , 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.

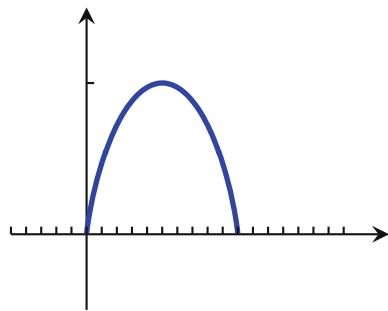
An attribute  $A$  with  $v$  possible values  $\{A_1, A_2, \dots, A_v\}$  creates a partition of the collection into  $v$  subsets, where each subset  $X_i$  corresponds to one value of  $A$  and contains  $p_i$  positive and  $n_i$  negative examples. The entropy of a subset is  $H(X_i)$  and the weighted average of entropies of the partition created by  $A$  is:

$$E(A) = \sum_{i=1}^v \frac{p_i + n_i}{p+n} H(X_i).$$

The information gain is defined as  $Gain(A) = H(X) - E(A)$  (or  $H_{\text{before}} - H_{\text{after}}$ ). We would reach the maximum possible gain with an attribute that creates

**Fig. 6.10** The binary entropy function:

$$-x \log_2 x - (1-x) \log_2 (1-x)$$



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. 6.9, let us compute the information gain of attribute *Outlook*. The entropy of the complete dataset is (Table 6.7):

$$H(X) = -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} = 0.940.$$

*Outlook* has three values: *sunny*, *overcast*, and *rain*. The respective subsets created by these values consist of the objects:

- sunny* : {1, 2, 8, 9, 11}, two positives and three negatives,
- overcast* : {3, 7, 12, 13}, all positives,
- rain* : {4, 5, 6, 10, 14}, three positives and two negatives,

and their entropies are:

$$\begin{aligned} \text{sunny} : H(X_1) &= -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} = 0.971. \\ \text{overcast} : H(X_2) &= 0. \\ \text{rain} : H(X_3) &= -\frac{3}{5} \log_2 \frac{3}{5} - \frac{2}{5} \log_2 \frac{2}{5} = 0.971. \end{aligned}$$

Thus

$$E(\text{Outlook}) = \frac{5}{14} H(X_1) + \frac{4}{14} H(X_2) + \frac{5}{14} H(X_3) = 0.694.$$

*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 dataset for all attributes, and the attribute with the highest gain:

$$A = \arg \max_{a \in SA} H(X) - E(a).$$

is selected to be the root of the tree. The dataset is then split into  $v$  subsets  $\{N_1, \dots, N_v\}$ , 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  $O(NM)$ .

### 6.2.4 Numerical Attributes

ID3 handles categorical attributes only. In a sequel to ID3, Quinlan (1993, p. 25) proposed a simple technique to deal with numerical, or continuous, attributes and find binary partitions for the corresponding nodes. Each partition is defined by a threshold, where the first part will be the values less than this threshold and the second one, the values greater. For each continuous attribute, the outline of the algorithm is:

1. First, sort the values observed in the dataset. There is a finite number of values:  $\{v_1, v_2, \dots, v_m\}$ ;
2. Then, for all the pairs of consecutive values  $v_i$  and  $v_{i+1}$ , compute the gain of the resulting split,  $\{v_1, v_2, \dots, v_i\}$  and  $\{v_{i+1}, \dots, v_m\}$ , and determine the pair,  $(v_{i_{max}}, v_{i_{max}+1})$ , which maximizes it. As threshold value, Quinlan (1993) proposes the midpoint:

$$\frac{v_{i_{max}} + v_{i_{max}+1}}{2}.$$

## 6.3 Encoding Categorical Values as Numerical Features

The dataset in Table 6.7 uses categorical—or nominal, or symbolic—attributes. While we can apply ID3 to this dataset, most classification methods, including other decision tree methods, will require numerical features as input. For all those classifiers, we need to convert the categorical attributes into numerical vectors before we can use them.

**Table 6.8** A representation of the categorical values in Table 6.7 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	<i>N</i>
2	1	0	0	1	0	0	1	0	1	0	<i>N</i>
3	0	1	0	1	0	0	1	0	0	1	<i>P</i>
4	0	0	1	0	1	0	1	0	0	1	<i>P</i>
5	0	0	1	0	0	1	0	1	0	1	<i>P</i>
6	0	0	1	0	0	1	0	1	1	0	<i>N</i>
7	0	1	0	0	0	1	0	1	1	0	<i>P</i>
8	1	0	0	0	1	0	1	0	0	1	<i>N</i>
9	1	0	0	0	0	1	0	1	0	1	<i>P</i>
10	0	0	1	0	1	0	0	1	0	1	<i>P</i>
11	1	0	0	0	1	0	0	1	1	0	<i>P</i>
12	0	1	0	0	1	0	1	0	1	0	<i>P</i>
13	0	1	0	1	0	0	0	1	0	1	<i>P</i>
14	0	0	1	0	1	0	1	0	1	0	<i>N</i>

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 (Suits 1957). Let us exemplify this with the *Outlook* attribute in Table 6.7:

- *Outlook* has three possible values:  $\{\text{sunny}, \text{overcast}, \text{rain}\}$ . 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. This corresponds to a unit vector. Using the examples in Table 6.7, 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 unit vector for each attribute. To represent a complete object, we will finally concatenate all these vectors into a larger one characterizing this object. Table 6.8 shows the complete conversion of the dataset using vectors of binary values.

This type of encoding is generally called **one-hot encoding**. This technique has also the names *dummy variables* or *indicator function*.

If an attribute has from the beginning a numerical value, it does not need to be converted.

## 6.4 Programming: Inducing Decision Trees with Scikit-Learn

scikit-learn (Pedregosa et al. 2011) is a popular and comprehensive machine-learning library that we will use in this book. It provides with a large set of supervised and unsupervised algorithms in Python. The `sklearn.tree` module, for instance, has a `DecisionTreeClassifier`, while `sklearn.linear_model` includes the perceptron and logistic regression, and `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:

- $\mathbf{x}$  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  $\mathbf{y}$ , the class vector for the whole dataset.

Both  $X$  and  $\mathbf{y}$  must be in the NumPy array format, see Chap. 5, *Python for Numerical Computations*, on which scikit-learn is built.

### 6.4.1 Conversion of Categorical Data

The scikit-learn's classifiers use numerical algorithms that we cannot apply to categorical data such as those in Table 6.7, where  $X$  and  $\mathbf{y}$  correspond to:

$$X = \begin{bmatrix} \text{Sunny} & \text{Hot} & \text{High} & \text{False} \\ \text{Sunny} & \text{Hot} & \text{High} & \text{True} \\ \text{Overcast} & \text{Hot} & \text{High} & \text{False} \\ \text{Rain} & \text{Mild} & \text{High} & \text{False} \\ \text{Rain} & \text{Cool} & \text{Normal} & \text{False} \\ \text{Rain} & \text{Cool} & \text{Normal} & \text{True} \\ \text{Overcast} & \text{Cool} & \text{Normal} & \text{True} \\ \text{Sunny} & \text{Mild} & \text{High} & \text{False} \\ \text{Sunny} & \text{Cool} & \text{Normal} & \text{False} \\ \text{Rain} & \text{Mild} & \text{Normal} & \text{False} \\ \text{Sunny} & \text{Mild} & \text{Normal} & \text{True} \\ \text{Overcast} & \text{Mild} & \text{High} & \text{True} \\ \text{Overcast} & \text{Hot} & \text{Normal} & \text{False} \\ \text{Rain} & \text{Mild} & \text{High} & \text{True} \end{bmatrix}; \mathbf{y} = \begin{bmatrix} N \\ N \\ P \\ P \\ P \\ N \\ P \\ N \\ P \\ N \end{bmatrix}$$

We need first to convert this dataset into binary vectors as we saw in Sect. 6.3. 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). As in Sect. 4.3, we 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_Xy(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_Xy(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.,  1.,  0.,  0.,  0.,  1.,  0.,  1.],
       [ 0.,  1.,  0.,  1.,  0.,  0.,  1.,  0.,  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 6.8. We set the `sparse` parameter to `False` to be able to visualize the matrix. In most cases, it should be set to `True` to save memory space.

We can then use any scikit-learn classifier to train a model and predict classes.

### 6.4.2 Inducing a Decision Tree with Scikit-Learn

scikit-learn has no implementation of ID3. Instead of it, scikit-learn's `DecisionTreeClassifier` provides a version of CART (Breiman et al. 1984), which is a similar algorithm and outputs binary decision trees.

We fit a model, here a decision tree with the entropy criterion as defined in Sect. 6.2.3, with the lines:

```
classifier = tree.DecisionTreeClassifier(criterion='entropy')
classifier.fit(X, y)
```

Once the model is trained, we can apply it to new observations. The next instruction just reapplyes it to the training set:

```
y_predicted = classifier.predict(X)
```

which predicts exactly the classes we had in this set:

```
array(['no', 'no', 'yes', 'yes', 'yes', 'no', 'yes', 'no',
       'yes', 'yes', 'yes', 'yes', 'yes', 'no'], dtype='|<U3')
```

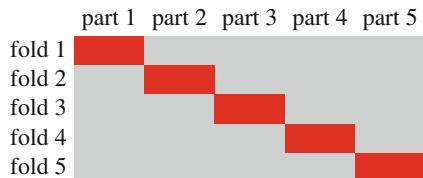
### 6.4.3 Evaluating a Model

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 7.1, 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.

In addition to the training and test sets, we often need a validation set also called a development set. Creating a model usually requires many training runs with different parameters and as many evaluations. As we saw, we should use the test set only once, in the final evaluation, otherwise the risk is to select algorithm parameters, for instance the criterion in the decision tree of Sect. 6.4.2, that are optimal for this test set. The validation set is an auxiliary test set that will help us select a model without touching the test set.

In the case of a small dataset like ours, a specific test set may lead to results that would be quite different with another test set. Cross validation, or  $N$ -fold cross validation, is a technique to mitigate such a bias. Instead of using one single test set, cross validation uses multiple splits called the folds. In a fivefold cross validation, the evaluation is carried out on five different test sets randomly sampled from the dataset. In each fold, the rest of the dataset serves as training set. Figure 6.11 shows a fivefold cross validation process, where we partitioned the dataset into five equal size subsets. In each fold, one of the subsets is the test set (red) and the rest, the

**Fig. 6.11** Fivefold cross validation with five different partitions. In each fold, the test set is in red and the training set in gray



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 fivefold 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())
```

In this run, we obtained a score of 0.7. Note that this figure may vary depending on the way scikit-learn splits the dataset.

Accuracy, the number of correctly classified observations, although very intuitive, is not always a good way to assess the performance of a model. Think, for example, of a dataset consisting of 99% positive observations and of 1% negative ones. A lazy classifier constantly choosing the positive class would have a 99% accuracy. We will see in Chap. 7, *Linear and Logistic Regression* more elaborate metrics to evaluate classification results.

## 6.5 Further Reading

Information theory is covered by many books, often requiring a good mathematical background. Manning and Schütze (1999, Chap. 2) provide 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. It may suffice in some application cases. Other more elaborate algorithms based on decision trees, such as random forests (Ho 1995) and gradient boosting (Friedman 2001), have a better performance. They are more complex and their presentation is out of the scope of this book.

A number of machine-learning toolkits feature programs to induce decision trees. We used scikit-learn (Pedregosa et al. 2011) to format data, induce a tree, and evaluate its performance. scikit-learn has an excellent documentation and all its code is available as open source.<sup>1</sup> R is another set of statistical and machine-learning functions<sup>2</sup> including classification algorithms based on decision trees. R uses its own script language. Weka (Witten and Frank 2005; Hall et al. 2009) is another collection of machine-learning algorithms with an implementation of ID3, this time written in Java.<sup>3</sup>

---

<sup>1</sup> <https://scikit-learn.org/>.

<sup>2</sup> <https://www.r-project.org/>.

<sup>3</sup> <https://www.cs.waikato.ac.nz/ml/weka/>.

# Chapter 7

## Linear and Logistic Regression



J'ay même trouvé une chose estonnante, c'est qu'on peut representer par les Nombres, toutes sortes de verit  s et consequences. [ . . . ] tous les raisonnemens se pourroient determiner    la fa  on des nombres, et m  mes    l'egard de ceux o   les circonstances donn  es, ou *data*, ne suffisent pas    la determination de la question, on pourroit neantmoins determiner mathematiquement le degr   de la probabilit  .

"I even found an astonishing thing, it is that we can represent by the numbers, all kinds of truths and consequences [ . . . ] all reasonings could be determined in the manner of numbers, and even with regard to those where the given circumstances, or *data*, are not sufficient for the determination of the question, one could nevertheless determine mathematically the degree of the probability."

G. W. Leibniz, *Projet et essais pour avancer l'art d'inventer*, 1688–1690, edition A VI 4 A, pp. 963–964. Translation: Google translate.

In this chapter, we will go on with the description of linear regression and linear classifiers among the most popular ones: the perceptron and logistic regression. We will notably outline the mathematical background and notation we need to use these techniques properly. In addition to being used as a stand-alone technique, logistic regression is a core component of most modern neural networks.

### 7.1 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 nominal values such as *sunny*, *hot*, and *high*.

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. 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 parameters used to characterize the objects.

## 7.2 Choosing a Dataset

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 parameters, 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 7.1 shows these counts in French and in English.

**Table 7.1** 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	# <i>A</i>	# Characters	# <i>A</i>
Chapter 1	36, 961	2503	35, 680	2217
Chapter 2	43, 621	2992	42, 514	2761
Chapter 3	15, 694	1042	15, 162	990
Chapter 4	36, 231	2487	35, 298	2274
Chapter 5	29, 945	2014	29, 800	1865
Chapter 6	40, 588	2805	40, 255	2606
Chapter 7	75, 255	5062	74, 532	4805
Chapter 8	37, 709	2643	37, 464	2396
Chapter 9	30, 899	2126	31, 030	1993
Chapter 10	25, 486	1784	24, 843	1627
Chapter 11	37, 497	2641	36, 172	2375
Chapter 12	40, 398	2766	39, 552	2560
Chapter 13	74, 105	5047	72, 545	4597
Chapter 14	76, 725	5312	75, 352	4871
Chapter 15	18, 317	1215	18, 031	1119
Total	619, 431	42, 439	608, 230	39, 056

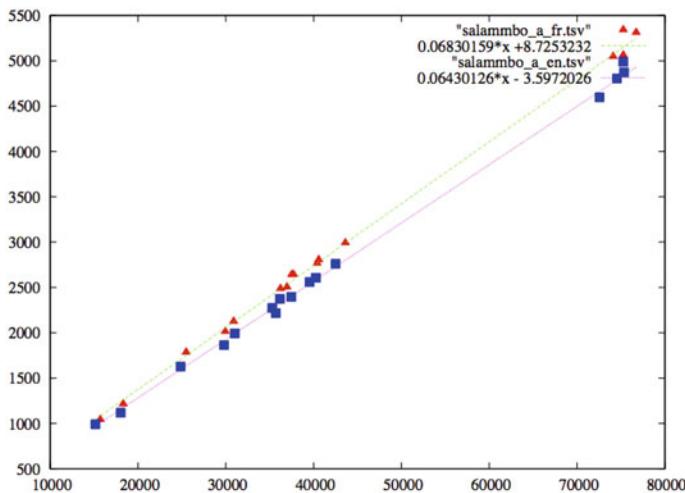
### 7.3 Linear Regression

Before we try to discriminate between French and English, let us examine how we can model the distribution of the letters in one language.

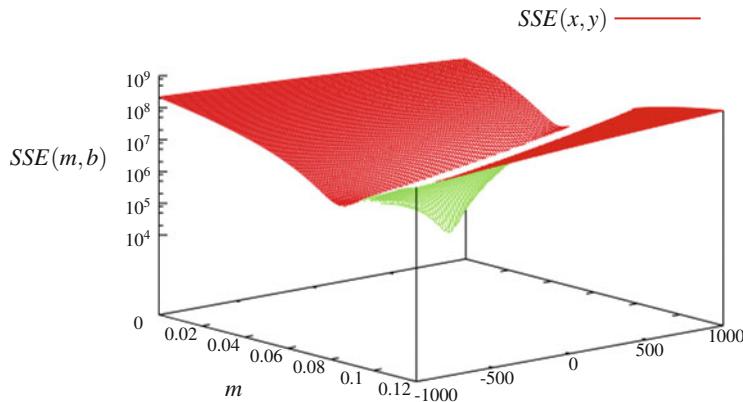
Figure 7.1 shows the plot of data in Table 7.1, where each point represents the letter counts in one of the 15 chapters. The  $x$ -axis corresponds to the total count of letters in the chapter, and the  $y$ -axis, the count of As. We can see from the figure that the points in both languages can be fitted 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  $\{(x_i, f(x_i)) | i = 1..q \text{ and } f(x_i) = mx_i + b\}$ . In our dataset, we have 15 observations from each chapter in *Salammbô*, and hence  $q = 15$ .



**Fig. 7.1** 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



**Fig. 7.2** Plot of  $SSE(m, b)$  applied to the 15 chapters of the English version of *Salammbo*

### 7.3.1 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):

$$\begin{aligned} SSE(m, b) &= \sum_{i=1}^q (f(x_i) - y_i)^2, \\ &= \sum_{i=1}^q (mx_i + b - y_i)^2. \end{aligned}$$

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 7.2 shows the plot of  $SSE(m, b)$  applied to the 15 chapters of the English version of *Salammbo*. 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:

$$\begin{aligned} \frac{\partial SSE(m, b)}{\partial m} &= \sum_{i=1}^q \frac{\partial}{\partial m} (mx_i + b - y_i)^2 = 2 \sum_{i=1}^q x_i (mx_i + b - y_i) = 0, \\ \frac{\partial SSE(m, b)}{\partial b} &= \sum_{i=1}^q \frac{\partial}{\partial b} (mx_i + b - y_i)^2 = 2 \sum_{i=1}^q (mx_i + b - y_i) = 0. \end{aligned}$$

We obtain then:

$$m = \frac{\sum_{i=1}^q x_i y_i - q \bar{x} \bar{y}}{\sum_{i=1}^q x_i^2 - q \bar{x}^2} \quad \text{and} \quad b = \bar{y} - m \bar{x},$$

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:

$$\text{French: } y = 0.0683x + 8.7253$$

$$\text{English: } y = 0.0643x - 3.5972$$

### 7.3.2 Least Absolute Deviation

As alternative loss function to the least squares, we can minimize the sum of the absolute errors (SAE) (Boscovich 1770, Livre V, note):

$$SAE(m, b) = \sum_{i=1}^q |f(x_i) - y_i|.$$

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.

## 7.4 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 As. 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 *Salammbo* example, the input would be the number of letters in a chapter, and the output, the number of As.

In a typical dataset such as the one shown in Table 7.1, 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  $(1, x_1, x_2, \dots, x_{n-1})$  or  $\mathbf{x}$ . In NumPy and PyTorch, we will represent them as a row vectors; see Sect. 5.5.8. The first parameter is set to 1 to take the intercept into account and make the computation easier. In the *Salammbo* example, this corresponds to the letter count in a chapter, for example:  $(1, 36, 961)$  in Chapter 1 in French. To denote the whole dataset, we use a matrix,  $X$ , where each row corresponds to an observation.

**The output value:** Each output, also called response or target, 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 *Salammbo*, in French, the count of As is  $y = 2503$  in Chapter 1, and the predicted value using the regression line is  $\hat{y} = 0.0683 \times 36,961 + 8.7253 = 2533.22$ . We use a vector,  $\mathbf{y}$ , to denote all the outputs in the dataset.

**The squared error:** The squared error is the squared difference between the prediction and the observed value,  $(\hat{y} - y)^2$ . In *Salammbo*, the squared error for Chapter 1 is  $(2533.22 - 2503)^2 = 30.22^2 = 913.26$ .

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  $(w_0, w_1, w_2, \dots, w_{n-1})$  or  $\mathbf{w}$ . These coefficients are usually called the weights. They correspond to  $(b, m)$  when  $n = 2$ . In *Salammbo*, 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  $\mathbf{w} = (w_0, w_1, w_2, \dots, w_{n-1})$ ,  $\mathbf{x} = (x_0, x_1, x_2, \dots, x_{n-1})$ , and  $x_0 = 1$ . Using a matrix notation, the predicted values,  $\hat{\mathbf{y}}$ , for all the observations in the dataset,  $X$ , are given by the product:

$$\hat{\mathbf{y}} = X\mathbf{w}.$$

For the French dataset, the complete matrix and vectors are:

$$X = \begin{bmatrix} 1 & 36961 \\ 1 & 43621 \\ 1 & 15694 \\ 1 & 36231 \\ 1 & 29945 \\ 1 & 40588 \\ 1 & 75255 \\ 1 & 37709 \\ 1 & 30899 \\ 1 & 25486 \\ 1 & 37497 \\ 1 & 40398 \\ 1 & 74105 \\ 1 & 76725 \\ 1 & 18317 \end{bmatrix}; \mathbf{w} = \begin{bmatrix} 8.7253 \\ 0.0683 \end{bmatrix}; \hat{\mathbf{y}} = \begin{bmatrix} 2533.22 \\ 2988.11 \\ 1080.65 \\ 2483.36 \\ 2054.02 \\ 2780.95 \\ 5148.76 \\ 2584.31 \\ 2119.18 \\ 1749.46 \\ 2569.83 \\ 2767.97 \\ 5070.21 \\ 5249.16 \\ 1259.81 \end{bmatrix}; \mathbf{y} = \begin{bmatrix} 2503 \\ 2992 \\ 1042 \\ 2487 \\ 2014 \\ 2805 \\ 5062 \\ 2643 \\ 2126 \\ 1784 \\ 2641 \\ 2766 \\ 5047 \\ 5312 \\ 1215 \end{bmatrix};$$

$$\mathbf{se} = \begin{bmatrix} 913.26 \\ 15.14 \\ 1493.86 \\ 13.25 \\ 1601.31 \\ 578.40 \\ 7527.51 \\ 3444.53 \\ 46.57 \\ 1193.04 \\ 5065.18 \\ 38920 \\ 538.909 \\ 3948.29 \\ 2007.53 \end{bmatrix}.$$

## 7.5 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 sections.

### 7.5.1 Mathematical Description

Gradient descent (Cauchy 1847) is a numerical method to find a global or local minimum of a function:

$$\begin{aligned} y &= f(w_0, w_1, w_2, \dots, w_n), \\ &= f(\mathbf{w}), \end{aligned}$$

even when there is no analytical solution.

As we can see on Fig. 7.2, the sum of squared errors has a minimum. This is 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  $(\mathbf{w}_k)$ , here representing the weight vectors, 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:

$$f(\mathbf{w}_1) > f(\mathbf{w}_2) > \dots > f(\mathbf{w}_k) > f(\mathbf{w}_{k+1}) > \dots > \min.$$

Now given a point, an initial weight vector,  $\mathbf{w}$ , 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  $\mathbf{w}$  by  $\mathbf{w} + \mathbf{v}$ , 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  $f(\mathbf{w}) > f(\mathbf{w} + \mathbf{v})$ .

As  $\|\mathbf{v}\|$  is small, we can approximate  $f(\mathbf{w} + \mathbf{v})$  using a Taylor expansion limited to the first derivatives:

$$f(\mathbf{w} + \mathbf{v}) \approx f(\mathbf{w}) + \mathbf{v} \cdot \nabla f(\mathbf{w}),$$

where the gradient of  $f$ , denoted  $\nabla f(\mathbf{w})$  and defined as:

$$\nabla f(w_0, w_1, w_2, \dots, w_n) = \left( \frac{\partial f}{\partial w_0}, \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \dots, \frac{\partial f}{\partial w_n} \right),$$

is a direction vector corresponding to the steepest slope.

We obtain the steepest descent when we choose  $\mathbf{v}$  collinear to  $\nabla f(\mathbf{w})$ :

$$\mathbf{v} = -\alpha \nabla f(\mathbf{w}), \text{ with } \alpha > 0.$$

We have then:

$$f(\mathbf{w} - \alpha \nabla f(\mathbf{w})) \approx f(\mathbf{w}) - \alpha \|\nabla f(\mathbf{w})\|^2,$$

and thus the inequality:

$$f(\mathbf{w}) > f(\mathbf{w} - \alpha \nabla f(\mathbf{w})).$$

This inequality enables us to write a recurrence relation between two consecutive steps:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla f(\mathbf{w}_k)$$

and find a step sequence to the minimum, where  $\alpha_k$  is a small positive number called the step size or learning rate. This number can be constant over all the descent or change at each step. The descent converges when the gradient becomes zero. This convergence is generally faster if the learning rate decreases or adapts to the gradient values over the iterations. In practice, we terminate the descent when  $\|\nabla f(\mathbf{w})\|$  is less than a predefined threshold, or is not decreasing, or has reached a maximum number of iterations.

Symmetrically to the steepest descent, we have a steepest ascent when

$$\mathbf{v} = \alpha \nabla f(\mathbf{w}).$$

When the function represents a loss that we will minimize, as with the sum of squared errors, we will denote it  $L(\mathbf{w})$  or  $Loss(\mathbf{w})$  and apply a descent. When it is a quantity to maximize, we will denote it  $\ell(\mathbf{w})$  and apply an ascent.

### 7.5.2 Gradient Descent and Linear Regression

For a dataset,  $DS$ , 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 dataset  $DS$  of  $q$  examples:  $DS = \{(1, x_{i,1}, y_i) | i : 1..q\}$ , where the error is defined as:

$$\begin{aligned} SSE(w_0, w_1) &= \sum_{i=1}^q (\hat{y}_i - y_i)^2, \\ &= \sum_{i=1}^q (w_0 + w_1 x_{i,1} - y_i)^2. \end{aligned}$$

The gradient of this two-dimensional equation  $\nabla SSE(\mathbf{w})$  is:

$$\begin{aligned}\frac{\partial SSE(w_0, w_1)}{\partial w_0} &= 2 \sum_{i=1}^q w_0 + w_1 x_{i,1} - y_i, \\ \frac{\partial SSE(w_0, w_1)}{\partial w_1} &= 2 \sum_{i=1}^q x_{i,1} \times (w_0 + w_1 x_{i,1} - y_i).\end{aligned}$$

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:

$$\begin{aligned}w_0 &\leftarrow w_0 - \frac{\alpha}{q} \cdot \sum_{i=1}^q w_0 + w_1 x_{i,1} - y_i, \\ w_1 &\leftarrow w_1 - \frac{\alpha}{q} \cdot \sum_{i=1}^q x_{i,1} \times (w_0 + w_1 x_{i,1} - y_i).\end{aligned}$$

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:

$$\begin{aligned}w_0 &\leftarrow w_0 - \alpha \cdot (w_0 + w_1 x_{i,1} - y_i) \\ w_1 &\leftarrow w_1 - \alpha \cdot x_{i,1} \cdot (w_0 + w_1 x_{i,1} - y_i).\end{aligned}$$

The examples are usually selected randomly from  $DS$ . This is called the **stochastic gradient descent or online learning**.

For large datasets, a batch gradient descent would be impractical as it would take too much memory. The stochastic variant has not this limitation and often has a faster convergence. It is more unstable however. To make the convergence more regular, most modern machine-learning toolkits use minibatches instead, where they compute the gradient by small subsets of 4 to 256 inputs. This technique is called the **minibatch gradient descent**.

The duration of the descent is measured in **epochs**, where an epoch is the period corresponding to one iteration over the complete dataset: the  $q$  examples.

## ***N*-Dimensional Space**

In an  $n$ -dimensional space, we want to find the regression hyperplane:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n,$$

given a dataset  $DS$  of  $q$  examples:  $DS = \{(1, x_{i,1}, x_{i,2}, \dots, x_{i,n}, y_i) | i : 1..q\}$ , where the error is defined as:

$$\begin{aligned} SSE(w_0, w_1, \dots, w_n) &= \sum_{i=1}^q (\hat{y}_i - y_i)^2, \\ &= \sum_{i=1}^q (w_0 + w_1 x_{i,1} + w_2 x_{i,2} + \dots + w_n x_{i,n} - y_i)^2. \end{aligned}$$

To simplify the computation of partial derivatives, we introduce the parameter  $x_{i,0} = 1$  so that:

$$SSE(w_0, w_1, \dots, w_n) = \sum_{i=1}^q (w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2} + \dots + w_n x_{i,n} - y_i)^2.$$

The gradient of  $SSE$  is defined by the partial derivatives:

$$\frac{\partial SSE}{\partial w_j} = 2 \sum_{i=1}^q x_{i,j} \cdot (w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2} + \dots + w_n x_{i,n} - y_i).$$

In the batch version, the iteration step considers all the examples in  $DS$ :

$$w_j \leftarrow w_j - \frac{\alpha}{q} \cdot \sum_{i=1}^q x_{i,j} \cdot (w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2} + \dots + w_n x_{i,n} - y_i).$$

or using a matrix notation:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{q} \cdot X^\top (\hat{\mathbf{y}} - \mathbf{y}),$$

where  $\hat{\mathbf{y}} = X\mathbf{w}$ .

In the stochastic version, we carry out the updates using one example at a time. For the  $i$ th example corresponding to  $\mathbf{x}_i$  and  $y_i$ , it results in the update:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot (\hat{y}_i - y_i) \cdot \mathbf{x}_i,$$

where  $\hat{y}_i = \mathbf{x}_i \cdot \mathbf{w}$ .

## 7.6 Regularization

### 7.6.1 The Analytical Solution Again

In Sect. 7.3.1, we saw that linear regression in a two-dimensional space had an analytical solution. This can be generalized for any dimension. Following the

formulation in Sect. 7.3.1, linear regression consists in finding the weight vector  $\mathbf{w}$  that minimizes the sum of squared errors between  $X\mathbf{w}$  and  $\mathbf{y}$ . Ideally, we would have this equality:

$$X\mathbf{w} = \mathbf{y}.$$

To solve this equation, we could think of inverting  $X$  and have:

$$\mathbf{w} = X^{-1}\mathbf{y}.$$

Unfortunately, this simple operation is not possible in general as  $X$  is not a square matrix and hence not invertible. Nonetheless, Fredholm (1903) and then Moore (1920) showed that we can define a **pseudoinverse** of  $X$  in the form of:

$$(X^\top X)^{-1}X^\top$$

that solves the least square problem:

$$\mathbf{w} = (X^\top X)^{-1}X^\top\mathbf{y}.$$

## 7.6.2 Inverting $X^\top X$

At this point, we may wonder if we can always derive a pseudoinverse. To do this, we need to invert  $X^\top X$ , which is possible if  $X$  has linearly independent columns. In real datasets, it is frequently the case that we have highly correlated columns or even duplicate ones. This means that  $X^\top X$  will be singular, and hence not invertible.

A way to make  $X^\top X$  invertible is to add it a scalar matrix  $\lambda I$ , where  $\lambda$  is small (Hoerl 1962):

$$\mathbf{w} = (X^\top X + \lambda I)^{-1}X^\top\mathbf{y}$$

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.

## 7.6.3 Regularization

Correlated features result in large values of  $\mathbf{w}$  that regularization tries to penalize. In most practical regression and classification cases, we replace the loss  $L$ , the sum of squared errors in regression, with a *Cost*:

$$Cost(\mathbf{w}) = L(\mathbf{w}) + \lambda L_q(\mathbf{w}),$$

where

$$L_q = \sum_{i=1}^n |w_i|^q.$$

The most frequent regularizations are:

$$L_2 = \sum_{i=1}^n w_i^2,$$

in what is called a ridge regression and:

$$L_1 = \sum_{i=1}^n |w_i|$$

in a LASSO regression. In practice,  $w_0$  is often part of the regularization.

## 7.7 Linear Classification

### 7.7.1 An Example

We will now use the dataset in Table 7.1 to describe classification techniques that split the texts into French or English. If we examine it closely, Fig. 7.1 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:

$$w_0 + w_1x + w_2y = 0$$

separating the plane into two half-planes defined by the inequalities:

$$w_0 + w_1x + w_2y > 0$$

and

$$w_0 + w_1x + w_2y < 0.$$

These inequalities mean that the points belonging to one class of the dataset are on one side of the separating line and the others are on the other side.

In Table 7.1 and Fig. 7.1, the chapters in French have a steeper slope than the corresponding ones in English. The points representing the French chapters will

**Table 7.2** Inequalities derived from Table 7.1 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	$1215 > w_0 + 18317w_1$	$1119 < w_0 + 18031w_1$

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:

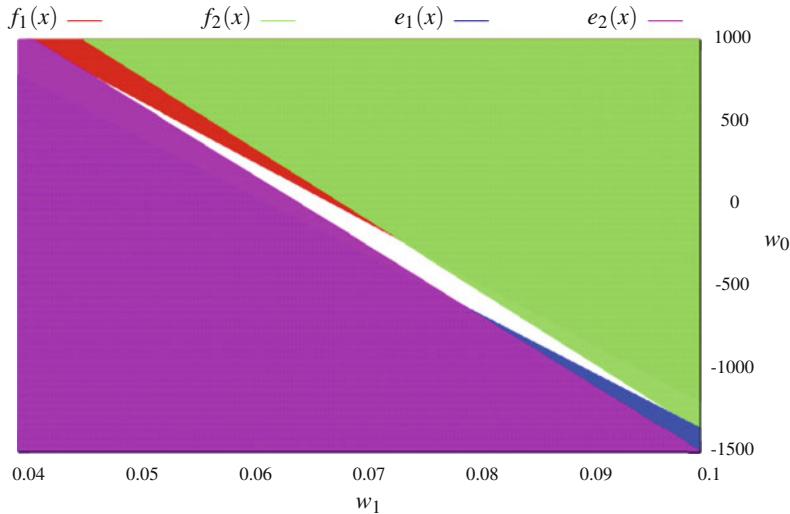
$$y_i > w_0 + w_1 x_i \text{ for the set of points: } \{(x_i, y_i) | (x_i, y_i) \in \text{French}\} \text{ and}$$

$$y_i < w_0 + w_1 x_i \text{ for the set of points: } \{(x_i, y_i) | (x_i, y_i) \in \text{English}\},$$

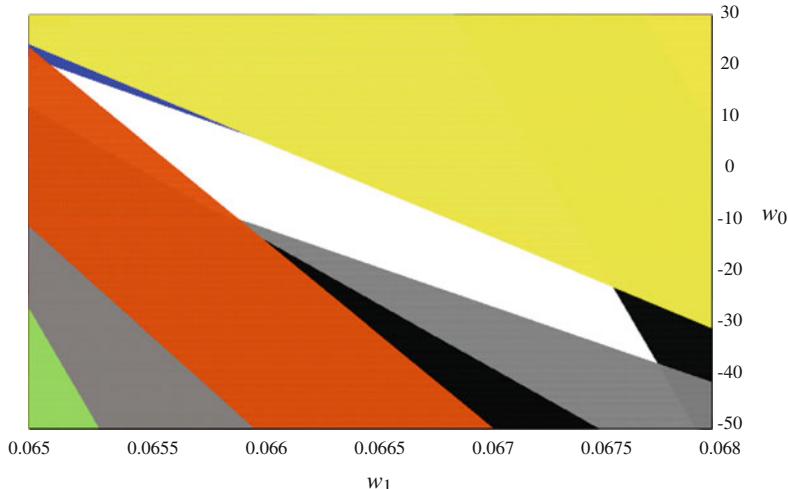
where  $x$  is the total count of letters in a chapter and  $y$ , the count of As. In total, we will have 30 inequalities, 15 for French and 15 for English shown in Table 7.2. Any weight vector  $\mathbf{w} = (w_0, w_1)$  that satisfies all of them will define a classifier correctly separating the chapters into two classes: French or English.

Let us represent graphically the inequalities in Table 7.2 and solve the system in the two-dimensional space defined by  $w_0$  and  $w_1$ . Figure 7.3 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 7.4 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.



**Fig. 7.3** 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



**Fig. 7.4** 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

### 7.7.2 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:

$$w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0$$

and

$$w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n < 0,$$

where each observation is described by a feature vector  $\mathbf{x}$ .

The sums in the inequalities correspond to the dot product of the weight vector,  $\mathbf{w}$ , by the the feature vector,  $\mathbf{x}$ , defined as

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^n w_i x_i,$$

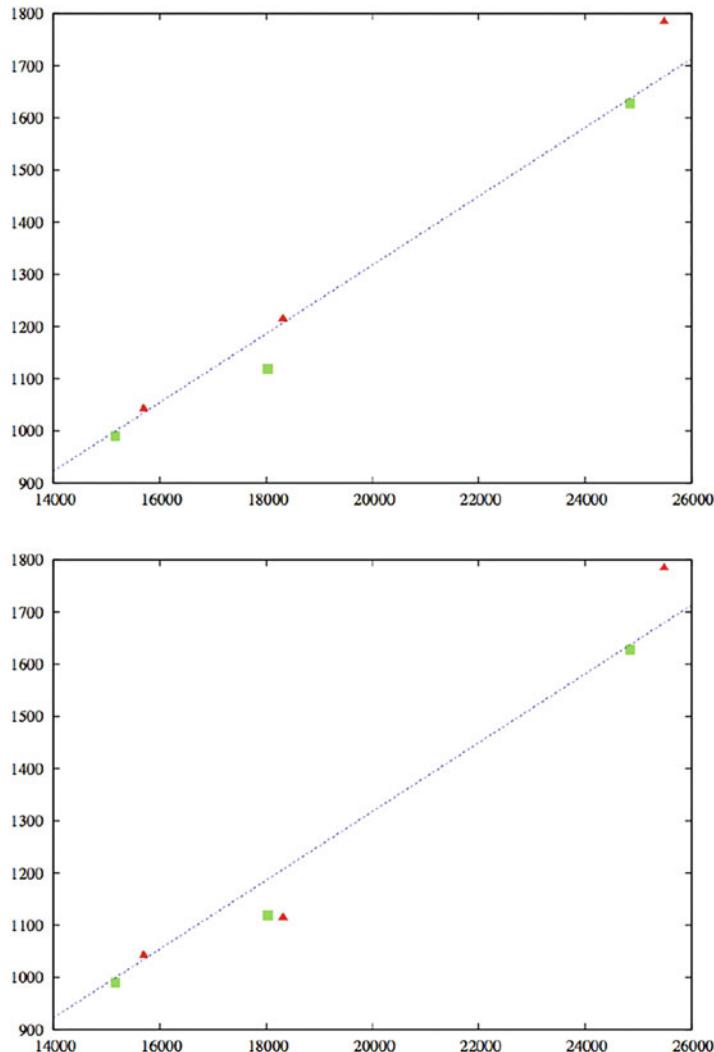
where  $\mathbf{w} = (w_0, w_1, w_2, \dots, w_n)$ ,  $\mathbf{x} = (x_0, x_1, x_2, \dots, x_n)$ , and  $x_0 = 1$ .

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. 10.3.2.

### 7.7.3 Linear Separability

It is not always the case that a line can perfectly separate the two classes of a dataset. Let us return to our dataset in Table 7.1 and restrict ourselves to the three shortest chapters: the 3rd, 10th, and 15th. Figure 7.5, 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 dataset, Chapter 10 in French has 18,317 letters and 1115 As instead of 18,317 and 1215, respectively. Figure 7.5, right, shows this plot. This time, no line can pass between the two classes, and the dataset 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.



**Fig. 7.5** *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

#### 7.7.4 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 As occurring in a text in French from the total number of characters.

Having 75,255 characters in this chapter, the regression line will predict 5149 occurrences of As (there are 5062 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 As in a text, classification will predict the language: French or English. For instance, having the pair (75,255, 5062), the classifier will predict French.

In the next sections, we will examine three categories of linear classifiers from among the most popular and efficient ones: perceptrons, logistic regression, and neural networks. For the sake of simplicity, we will first 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. This is the most frequent case in practice and we will then see how to apply logistic regression to multinomial cases.

## 7.8 Perceptron

Given a dataset like the one in Table 7.1, 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  $(w_0, w_1, w_2, \dots, w_n)$  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.

### 7.8.1 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 dataset :  $DS = \{(1, x_{i,1}, x_{i,2}, \dots, x_{i,n}, y_i) | i : 1..q\}$  of  $q$  examples, where  $y_i \in \{0, 1\}$ , we have:

$$\begin{aligned}\hat{y}(\mathbf{x}_i) &= H(\mathbf{w} \cdot \mathbf{x}_i), \\ &= H(w_0 + w_1 x_{i,1} + w_2 x_{i,2} + \dots + w_n x_{i,n}).\end{aligned}$$

We use  $x_{i,0} = 1$  to simplify the equations and the set  $\{0, 1\}$  corresponds to the classes {English, French} in Table 7.1.

### 7.8.2 The Iteration

Let us denote  $\mathbf{w}_k$  the weight vector at step  $k$ . 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  $\mathbf{w}_k \cdot \mathbf{x}_i$  one object at a time for all the members of the dataset,  $i : 1..q$ :

- 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:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \cdot (\hat{y}_i - y_i) \cdot \mathbf{x}_i,$$

until all the objects are correctly classified.

For a misclassified object, we have  $\hat{y}_i - y_i$  equals to either  $1 - 0$  or  $0 - 1$ . The update value is then is  $-\alpha \cdot \mathbf{x}_i$  or  $\alpha \cdot \mathbf{x}_i$ , where  $\alpha$  is the learning rate. For an object that is correctly classified, we have  $\hat{y}_i - y_i = 0$ , 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:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\alpha}{q} X^T (\hat{\mathbf{y}} - \mathbf{y}).$$

### 7.8.3 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:  $\mathbf{x}_i = (1, x_{i,1}, x_{i,2})$  and  $\mathbf{w} = (w_0, w_1, w_2)$ . With the stochastic gradient descent, we carry out the updates using the relations:

$$\begin{aligned} w_0 &\leftarrow w_0 - (\hat{y}_i - y_i) \cdot 1, \\ w_1 &\leftarrow w_1 - (\hat{y}_i - y_i) \cdot x_{i,1}, \\ w_2 &\leftarrow w_2 - (\hat{y}_i - y_i) \cdot x_{i,2}, \end{aligned}$$

where  $\hat{y}_i - y_i$  is either, 0,  $-1$ , or  $1$ .

### 7.8.4 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 dataset.

## 7.9 Logistic Regression

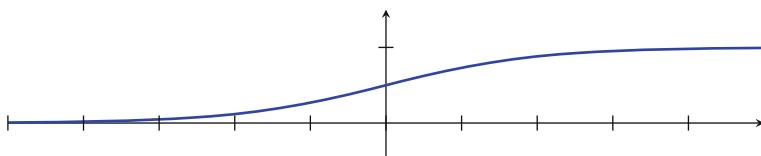
In its elementary formulation, the perceptron uses hyperplanes as absolute, unmitigated boundaries between the classes. In many datasets, however, there are no such clear-cut thresholds to separate the points. Figure 7.5 is an example of this that shows regions where the nonlinearly separable classes 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  $\mathbf{x}$  (an observation) to belong to a class. Figure 7.6 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. 7.6, where the



**Fig. 7.6** The logistic curve:  $f(x) = \frac{1}{1 + e^{-x}}$

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.

### 7.9.1 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 dataset,  $DS = \{(1, x_{i,1}, x_{i,2}, \dots, x_{i,n}, y_i) | i : 1..q\}$ , 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 dataset is:

$$\begin{aligned} \mathcal{L}(\mathbf{w}) &= \prod_{\mathbf{x}_i \in P} P(y_i = 1|\mathbf{x}_i) \times \prod_{\mathbf{x}_i \in N} P(y_i = 0|\mathbf{x}_i), \\ &= \prod_{\mathbf{x}_i \in P} P(y_i = 1|\mathbf{x}_i) \times \prod_{\mathbf{x}_i \in N} (1 - P(y_i = 1|\mathbf{x}_i)). \end{aligned}$$

We can rewrite the product using  $y_i$  as powers of the probabilities as  $y_i = 0$ , when  $\mathbf{x}_i \in N$  and  $y_i = 1$ , when  $\mathbf{x}_i \in P$ :

$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= \prod_{\mathbf{x}_i \in P} P(y_i = 1|\mathbf{x}_i)^{y_i} \times \prod_{\mathbf{x}_i \in N} (1 - P(y_i = 1|\mathbf{x}_i))^{1-y_i}, \\ &= \prod_{(\mathbf{x}_i, y_i) \in DS} P(y_i = 1|\mathbf{x}_i)^{y_i} \times (1 - P(y_i = 1|\mathbf{x}_i))^{1-y_i}.\end{aligned}$$

### Maximizing the Likelihood

We fit  $\mathbf{w}$ , and train a model by maximizing the likelihood of the observed classification:

$$\hat{\mathbf{w}} = \arg \max_{\mathbf{w}} \prod_{(\mathbf{x}_i, y_i) \in DS} P(y_i = 1|\mathbf{x}_i)^{y_i} \times (1 - P(y_i = 1|\mathbf{x}_i))^{1-y_i}.$$

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**:

$$\hat{\mathbf{w}} = \arg \max_{\mathbf{w}} \sum_{(\mathbf{x}_i, y_i) \in DS} y_i \ln P(y_i = 1|\mathbf{x}_i) + (1 - y_i) \ln(1 - P(y_i = 1|\mathbf{x}_i)).$$

Using the logistic functions to express the probabilities, we have:

$$\hat{\mathbf{w}} = \arg \max_{\mathbf{w}} \sum_{(\mathbf{x}_i, y_i) \in DS} y_i \ln \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}_i}} + (1 - y_i) \ln \frac{e^{-\mathbf{w} \cdot \mathbf{x}_i}}{1 + e^{-\mathbf{w} \cdot \mathbf{x}_i}}.$$

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.

#### 7.9.2 Gradient Ascent ...

We can use gradient ascent to compute the maximum of the log-likelihood. This method is analogous to gradient descent that we saw in Sect. 7.5; we move upward instead. A Taylor expansion of the log-likelihood gives us:  $\ell(\mathbf{w} + \mathbf{v}) = \ell(\mathbf{w}) + \mathbf{v} \cdot \nabla \ell(\mathbf{w}) + \dots$ . 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-likelihood. 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.

### ...and Descent

Alternatively, we could try to find a minimum for the negative log-likelihood (NLL). We call the corresponding function the **logistic loss**, **log-loss**, or **binary cross-entropy**. For one observation, it is defined as:

$$L(\hat{y}, y) = -y \ln \hat{y} - (1 - y) \ln(1 - \hat{y}),$$

where

$$\hat{y} = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}.$$

We will then use a gradient descent to find the optimal parameters as in Sect. 7.5. This is how logistic regression is implemented in practice, where we minimize the binary cross-entropy. In addition, differently to Sect. 6.1.4, instead of the binary logarithm, most machine learning practitioners use the natural logarithm and compute the mean for the dataset:

$$BCELoss(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{q} \sum_{i=1}^q y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i),$$

where

$$\hat{y}_i = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}_i}}.$$

### Computing the Gradient

To compute the gradient of  $\nabla_{\mathbf{w}} BCELoss(\hat{\mathbf{y}}, \mathbf{y})$ , we will consider the loss for one point  $(\mathbf{x}_i, y_i)$ :

$$L(\hat{y}_i, y_i) = -y_i \ln \hat{y}_i - (1 - y_i) \ln(1 - \hat{y}_i).$$

The gradient of the loss  $\nabla_{\mathbf{w}} L(\hat{y}, y)$  is defined by the partial derivatives:  $\frac{\partial L(\hat{y}_i, y_i)}{\partial w_j}$ . Using the chain rule, we have:

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial w_j} = \frac{dL(\hat{y}_i, y_i)}{d\hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_j}.$$

Let us compute separately the two terms to the right of this equality, first  $\frac{dL(\hat{y}_i, y_i)}{d\hat{y}_i}$  and then  $\frac{\partial \hat{y}_i}{\partial w_j}$ :

1. The derivative of the loss with respect to  $\hat{y}_i$  is:

$$\begin{aligned}\frac{dL(\hat{y}_i, y_i)}{d\hat{y}_i} &= \frac{d}{d\hat{y}_i} (-y_i \ln \hat{y}_i - (1 - y_i) \ln(1 - \hat{y}_i)), \\ &= -\frac{y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i}, \\ &= \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)}.\end{aligned}$$

2. For the second term, using the chain rule again, we have:

$$\frac{\partial \hat{y}_i}{\partial w_j} = \frac{d\hat{y}_i}{d\mathbf{w} \cdot \mathbf{x}_i} \cdot \frac{\partial \mathbf{w} \cdot \mathbf{x}_i}{\partial w_j}.$$

Let us compute the two terms of the product to the right:

(a) The first term is the derivative of the logistic function:

$$\begin{aligned}\left(\frac{1}{1+e^{-x}}\right)' &= \frac{e^{-x}}{(1+e^{-x})^2}, \\ &= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right)\end{aligned}$$

We have then:

$$\frac{d\hat{y}_i}{d\mathbf{w} \cdot \mathbf{x}_i} = \hat{y}_i \cdot (1 - \hat{y}_i),$$

which is a useful identity.

(b) The partial derivative of  $\frac{\partial \mathbf{w} \cdot \mathbf{x}_i}{\partial w_j}$  is simply  $x_{i,j}$ .

We multiply these two terms and we have:

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial w_j} &= \frac{d\hat{y}_i}{d\mathbf{w} \cdot \mathbf{x}_i} \cdot \frac{\partial \mathbf{w} \cdot \mathbf{x}_i}{\partial w_j}, \\ &= \hat{y}_i \cdot (1 - \hat{y}_i) \cdot x_{i,j}.\end{aligned}$$

Finally, we can rewrite the gradient parameters as:

$$\begin{aligned}\frac{\partial L(\hat{y}_i, y_i)}{\partial w_j} &= \frac{\text{d}L(\hat{y}_i, y_i)}{\text{d}\hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_j}, \\ &= \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \cdot \hat{y}_i \cdot (1 - \hat{y}_i) \cdot x_{i,j}, \\ &= (\hat{y}_i - y_i) \cdot x_{i,j}.\end{aligned}$$

## 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  $DS = \{(1, x_{i,1}, x_{i,2}, \dots, x_{i,n}, y_i) | i : 1..q\}$ , the updates of  $\mathbf{w} = (w_0, w_1, \dots, w_n)$  are:

- With the stochastic gradient descent:

$$w_{j(k+1)} = w_{j(k)} - \alpha \cdot \left( \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}_i}} - y_i \right) \cdot x_{i,j};$$

or for the whole  $\mathbf{w}$  vector:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha \cdot (\hat{y}_i - y_i) \cdot \mathbf{x}_i;$$

- With the batch gradient descent:

$$w_{j(k+1)} = w_{j(k)} - \frac{\alpha}{q} \cdot \sum_{i=1}^q x_{i,j} \cdot \left( \frac{1}{1 + e^{-\mathbf{w}_k \cdot \mathbf{x}_i}} - y_i \right)$$

or using a matrix notation:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\alpha}{q} X^\top (\hat{\mathbf{y}} - \mathbf{y}).$$

We stop the descent when the gradient is less than a predefined threshold or after a certain number of epochs.

## 7.10 Gradient Descent Optimization

In this chapter so far, when fitting our models, we used constant learning rates in the update rules. The value of such learning rates has a considerable influence on the final results. A low rate will make the descent converge slowly while a high rate may overshoot the minimum. One possible method to find an optimal value is to

carry out convergence experiments on a part of the dataset with different rates. It is a common practice to vary the  $\alpha$  values between 0.1 and  $10^5$ , and look at the loss function with respect to the epochs.

Another way to optimize gradient descent is to use an adaptive learning rate that changes with the epochs. Such optimizers include Momentum (Qian 1999), RMSProp (Hinton 2012), Adam (Kingma and Ba 2014), and NAdam (Dozat 2016). We examine here the Momentum and RMSProp optimizers:

### 7.10.1 The Momentum

Qian (1999) noticed when the loss surface was a long and narrow valley, the descent trajectory oscillated between the ridges making it particularly slow. He proposed to redefine the update rule so that it could cancel the oscillations and have a more direct descent.

He added a momentum representing the accumulated past gradients to the update term  $\alpha \nabla_{\mathbf{w}} L(\mathbf{w})$ . The new update term is:

$$\Delta_k = \rho \Delta_{k-1} + \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_k) \text{ if } k > 0,$$

where  $\rho$  is the momentum parameter, for instance 0.9, with the initialization:

$$\Delta_0 = \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_0).$$

The momentum update rule becomes then:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \Delta_k.$$

### 7.10.2 RMSprop

RMSprop starts from the update rule of the gradient descent applied to the loss function as we defined it in Sect. 7.5:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla L(\mathbf{w}_k),$$

but it divides the  $\alpha$  constant by a moving average of the squared gradient.

The squared gradient is a scalar and its value at step  $k$  is  $\nabla L(\mathbf{w}_k)^2$ . The moving average is the weighted mean of the squared gradient and the previous moving average. Let us denote  $ms$  this moving average. We have:

$$ms_k = \rho ms_{k-1} + (1 - \rho) \nabla L(\mathbf{w}_k)^2,$$

and the following update rule:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\alpha}{\sqrt{ms_k}} \nabla L(\mathbf{w}_k).$$

In his slide, Hinton (2012) used a  $\rho$  value of 0.9.

Finally, Adam and NAdam are also adaptive learning rates. Starting from RMSprop, Adam replaces the gradient in the update rule with its moving average. NAdam modifies Adam further and adds a momentum to the gradient. Adam and NAdam usually show a better convergence than RMSprop. They also reduce the sensitivity to the size of the mini-batch.

## 7.11 Programming Logistic Regression with Scikit-Learn

In this section, we will apply logistic regression to our small *Salammbô* dataset with the scikit-learn toolkit. We already used scikit-learn in Sect. 6.4 and we saw that its base numerical representation was NumPy arrays (Chap. 5, *Python for Numerical Computations*). We will load and format the dataset in NumPy, fit a model, predict classes, and evaluate the performances with the scikit-learn API.

### **7.11.1 Representing the Dataset**

All the feature values in the dataset in Table 7.1 are numeric and creating  $X$  and  $y$  in a NumPy array format is straightforward. We can do it manually as it is a very small dataset. 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

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]
    ])
y = np.array(
    [0, 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, 1])
```

### 7.11.2 Scaling the Data

In the programs in this chapter, we will use the  $X$  values as is. However, a common practice is to scale them so that they range from 0 to 1 or from  $-1$  to  $+1$ . This will considerably improve the stability of the gradient descent. We will see these techniques in the next chapter (Sect. 8.5.1) as well as the corresponding scikit-learn functions.

### 7.11.3 Loading the Dataset from a File

In most applications, the dataset will be larger and we will have to load it from a file. There are many possible formats and loader applications. We give two examples here with tab-separated values (TSV) and svmlight.

#### Loading a TSV File with Pandas

In Sect. 4.3, we already used the TSV format and pandas. For our dataset, a TSV file will consist of three columns containing the number of characters, the number of As, and the class:

```
35680 2217 0
42514 2761 0
15162 990 0
...
36961 2503 1
43621 2992 1
15694 1042 1
...
```

We load the dataset in a panda `DataFrame` with the statements:

```
import pandas as pd

dataset_pd = pd.read_csv('../salammbo/salammbo_a_binary.tsv',
                        sep='\t',
                        names=['cnt_chars', 'cnt_a', 'class'])
```

and we convert it to numpy arrays with:

```
X = dataset_pd.to_numpy()[:, :2]
y = dataset_pd.to_numpy()[:, 2]
```

## The Svmlight Format

svmlight is an older format, but still widely used to distribute numerical datasets. Each row has the structure:

```
<class-label> <feature-idx>:<value> <feature-idx>:<value> ...
```

For the dataset in Table 7.1, 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
...
```

When a feature value is 0, we do not need to store it. This means that svmlight is well suited when the data is sparse.

We load our dataset with this piece of code:

```
from sklearn.datasets import load_svmlight_file
X, y = load_svmlight_file(file)
```

### 7.11.4 Fitting a Model with Scikit-Learn

Once we have our dataset ready in NumPy arrays, we select and fit a model, here logistic regression with default parameters, with the lines:

```
from sklearn.linear_model import LogisticRegression
classifier = linear_model.LogisticRegression()
classifier.fit(X, y)
```

Depending on the size of the dataset, the model can take a while to train. Here it is instantaneous. When the fitting is done, we can predict the class of new observations:

```
classifier.predict([X[-1]])          # 1
classifier.predict(np.array([[35680, 2217]])) # 0
```

The next instruction reapplies the model to the whole 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]
```

A more thorough evaluation would use a test set or cross-validation. This is what we will do in Sect. 7.12, but first let us examine the model.

### 7.11.5 The Logistic Regression Model

We saw in Sect. 7.9 that a logistic curve models the probabilities of a prediction: We obtain them with `predict_proba()`, which returns two values:  $P(0|\mathbf{x})$  and  $P(1|\mathbf{x})$ :

```
classifier.predict_proba([X[2]]) # [[0.9913024, 0.0086976]]
classifier.predict_proba([X[-1]]) # [[0.0180183, 0.9819817]]
```

The model itself consists of a weight vector  $(w_1, w_2)$  and an intercept  $w_0$ , respectively `coef_` and `intercept_`:

```
classifier.coef_          # [[-0.03372363 0.51169867]]
```

and

```
classifier.intercept_     # [-4.51879339e-05]
```

Predicting the probability of a class is just the application of the logistic function to the dot product  $\mathbf{w} \cdot \mathbf{x}$ . For this, we create the weight and feature vectors, respectively  $\mathbf{w} = (w_0, w_1, w_2)$  and  $\mathbf{x} = (1, x_1, x_2)$ , here for  $\mathbf{x}[-1]$ :

```
w = np.append(classifier.intercept_, classifier.coef_)
x = np.append([1.0], X[-1])
```

and we apply the logistic function:

```
1/(1 + np.exp(-w @ x))    # 0.9819817031873619
```

that returns the same value as with `predict_proba`.

Note that as the initial weight vectors are randomly initialized, the model parameters will probably differ between two fitting experiments. This implies that the results and figures shown in this book will certainly be slightly different in your own experiments.

### 7.11.6 The Loss

In Sect. 7.9.1, we saw that the binary cross-entropy loss defined by

$$BCELoss(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{q} \sum_{i=1}^q y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)$$

enabled us to fit the model. This may seem abstract and, given a model and a dataset, how do we compute it concretely?

In the equation, each pair  $(\hat{y}_i, y_i)$  represents an observation, where  $y_i$  is the class and  $\hat{y}_i$  the probability it belongs to class 1. The value of  $y_i$  is either 0 or 1 and  $\hat{y}_i$  is a number ranging from 0 to 1, for instance:

- The third observation,  $\mathbf{x}[2]$ , belongs to class 0 and we have

$$(\hat{y}_2, y_2) = (0.0086976, 0);$$

- The last one,  $x[-1]$ , belongs to class 1 and we have

$$(\hat{y}_{29}, y_{29}) = (0.9819817, 1).$$

We plug these values in the equation to obtain the loss for  $x[2]$  and  $x[-1]$ :

$$\begin{aligned} BCELoss(\hat{y}_2, y_2) &= -0 \cdot \ln(0.0086976) - (1 - 0) \cdot \ln(1 - 0.0086976), \\ &= -\ln(0.9913024), \\ &= 0.0087, \\ BCELoss(\hat{y}_{29}, y_{29}) &= -1 \cdot \ln(0.9819817) - (1 - 1) \ln(1 - 0.9819817), \\ &= -\ln(0.9819817), \\ &= 0.0182. \end{aligned}$$

To apply this equation the full dataset, we extract the probabilities of class 1 with:

```
classifier.predict_proba(X)[:, 1]
```

and we compute the loss with the loop:

```
-np.mean(np.log([y_hat if y_obs else 1 - y_hat
                 for y_obs, y_hat in
                 zip(y, classifier.predict_proba(X)[:, 1])]))
# 0.00206177
```

scikit-learn's has a dedicated built-in function for this and in routine programming, we will probably prefer to use it:

```
from sklearn import metrics

metrics.log_loss(y, classifier.predict_proba(X))
# 0.00206177
```

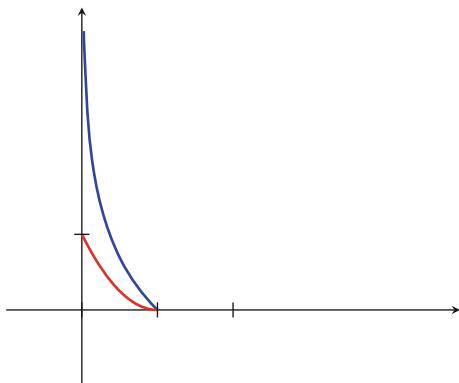
### 7.11.7 Comparing Cross Entropy and the Squared Error

In the linear regression algorithm in Sect. 7.3, we used a loss consisting of the sum or mean of squared errors (MSE), while in logistic regression, we used binary cross entropy and the probability logarithms. Binary cross entropy is a consequence of our decision to reach the maximum likelihood in Sect. 7.9.1.

In fact, we could also measure the prediction loss as the distance between the predicted probability and the value of the true class, either 0 or 1. Figure 7.7 shows these losses relatively to class 1 and compares the binary cross entropy loss defined as

$$BCELoss(\hat{y}) = -\log \hat{y}$$

**Fig. 7.7** Comparing the logistic loss in blue and the squared error loss in red



in blue with the squared error loss in red

$$MSELoss(\hat{y}) = (1 - \hat{y})^2.$$

Computing the squared error on our previous examples, we would have:

$$\begin{aligned} MSELoss(\hat{y}_2, y_2) &= (0 - 0.0086976)^2, \\ &= 7.56 \cdot 10^{-5}, \\ MSELoss(\hat{y}_{29}, y_{29}) &= (1 - 0.9819817)^2, \\ &= 3.25 \cdot 10^{-4}. \end{aligned}$$

and, as the figures show, this would be a bad idea. The squared error applied to probabilities leads to much smaller differences between the truth and the prediction. This makes it more difficult for a gradient descent to find a minimum. Figure 7.7 shows that the squared error squashed the loss range, while the negative logarithm strongly penalizes probability errors. A system that would predict with certainty class 0 when the true class is 1 would get an infinite binary cross entropy loss.

Although Berkson (1944) used squared errors in his paper on logistic regression, we should avoid this loss and always prefer cross entropy.

### 7.11.8 Multinomial Logistic Regression

Logistic regression extends to more than two classes. We call it a multinomial classification then. With scikit-learn, we use exactly the same functions. To illustrate it, let us extract statistics from the German translation of *Salammbo* from Project Gutenberg<sup>1</sup> and train a classifier for three languages.

---

<sup>1</sup> <https://www.gutenberg.org/>.

```
X_de = np.array(
    [[37599, 1771], [44565, 2116], [16156, 715], [37697, 1804],
     [29800, 1865], [42606, 2146], [78242, 3813], [40341, 1955],
     [31030, 1993], [26676, 1346], [39250, 1902], [41780, 2106],
     [72545, 4597], [79195, 3988], [19020, 928]
    ])
```

We add this array to  $X$  with `vstack` and we create a new  $y$  vector where class 2 is German:

```
X = np.vstack((X, X_de))
y = np.array(
    [0, 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, 1,
     2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

We create a new classifier, we fit it, and we predict the three classes of the training set with the same statements as with the binary classification:

```
cls_de = LogisticRegression()
cls_de.fit(X, y)
y_hat = cls_de.predict(X)

# array([2, 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, 1,
       2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2])
```

With this experiment, we saw that scikit-learn's programming interface is identical for binary and multinomial logistic regression. Nonetheless, the probability model and loss behind them are not exactly the same. We will see how to generalize logistic regression in Sect. 8.6.

## 7.12 Evaluation of Classification Systems

### 7.12.1 Accuracy

The prediction this time is not as perfect as with English and French. We can evaluate it more precisely with scikit-learn built-in functions using a  $N$ -fold cross-validation with  $N = 5$  and first with accuracy: How many observations the classifier predicted correctly.

We use a scikit-learn function again that shuffles the dataset, splits it  $N$  times into training and test sets, fits the models, and evaluates them. The function stratifies the data, meaning that the splits keep the original percentage of observations for each class. Each split uses about 80% of the data for the training set and the rest for the test set.

```

from sklearn.model_selection import cross_val_score

scores = cross_val_score(cls_de, X, y, cv=5, scoring='accuracy')
# array([0.88888889, 0.88888889, 0.88888889, 1.           ,
       0.88888889])
scores.mean()
# 0.91

```

### 7.12.2 Precision and Recall

Accuracy is sometimes misleading as we discussed in Sect. 6.4.3. A better evaluation of a classification result is to report three figures per class: recall, precision, and the *F*-measure. This latter metric, originally borrowed from library science, proved very generic to summarize the overall effectiveness of a system. It has been used in many other fields of language processing since then.

To explain these figures, let us stay in our library and imagine we want to retrieve all the documents on a specific topic, say *morphological parsing*. An automatic system to query the library catalog will, we hope, return some of them, but possibly not all. On the other hand, anyone who has searched a catalog knows that we will get irrelevant documents: *morphological pathology*, *cell morphology*, and so on. Table 7.3 summarizes the possible cases into which documents fall.

**Recall** measures how much relevant information the system has retrieved. It is defined as the number of relevant documents retrieved by the system divided by number of relevant documents in the library:

$$\text{Recall} = \frac{|A|}{|A \cup C|}.$$

For a given class, this would correspond to how many observations have been predicted correctly to be in this class with respect to their true number in the dataset.

**Precision** is the accuracy of what has been returned. It measures how much of the information is actually correct. It is defined as the number of correct documents returned divided by the total number of documents returned.

$$\text{Precision} = \frac{|A|}{|A \cup B|}.$$

For a given class, this would translate as: Out of the observations predicted to be in this class, how many are correct?

**Table 7.3** Documents in a library returned from a catalog query and split into relevant and irrelevant books

	Relevant documents	Irrelevant documents
Retrieved	<i>A</i>	<i>B</i>
Not retrieved	<i>C</i>	<i>D</i>

Recall and precision are combined into the ***F*-measure**, which is defined as the harmonic mean of both numbers:

$$F = \frac{2}{\frac{1}{P} + \frac{1}{R}} = \frac{2PR}{P+R}.$$

The *F*-measure is a composite metric that reflects the general performance of a system. It does not privilege precision at the expense of recall, or vice versa. An arithmetic mean would have made it very easy to reach 50% using, for example, very selective rules with a precision of 100 and a low recall.

Using a  $\beta$ -coefficient, it is possible to give an extra weight to either precision,  $\beta > 1$ , or recall,  $\beta < 1$ , however:

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}.$$

Finally, a **fallout** figure is also sometimes used that measures the proportion of irrelevant documents that have been selected.

$$\text{Fallout} = \frac{|B|}{|B \cup D|}.$$

Coming back to our *Salammbô* experiment and the prediction on the training set, we print the scores for the three classes with scikit-learn's classification report:

```
print(metrics.classification_report(y, y_hat))
```

	precision	recall	f1-score	support
0	0.81	0.87	0.84	15
1	1.00	1.00	1.00	15
2	0.86	0.80	0.83	15
accuracy			0.89	45
macro avg	0.89	0.89	0.89	45
weighted avg	0.89	0.89	0.89	45

where the macro average is the arithmetic mean of the F-1 scores. This macro average is frequently used as it synthesizes the performance of classification systems in just one number.

## 7.13 Further Reading

Logistic regression and perceptrons are popular classifiers, in addition to other techniques such as support-vector machines. Which one to choose has no easy answer as they may have different performances on different datasets. Nonetheless,

when starting an analysis, I would recommend logistic regression as 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 (2022), James et al. (2021), and Goodfellow et al. (2016) are mathematical references on classification and statistical learning in general that can complement this chapter. Grus (2019) is a pedagogical description on how to implement machine-learning algorithms in Python.

Gradient descent is the core algorithm to fit machine learning models. It has many variants and optimizations. We examined a few of them in this chapter. Ruder (2017) gives a very good overview of the most popular ones as well as practical optimizing strategies.

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 dataset 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. 7.1, 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. The algorithms also often extend to sequences of two and three characters. The Compact Language Detector v3 (CLD3)<sup>2</sup> is an example of modern language detector that uses a simple feed-forward neural network with one hidden layer. We will see an implementation of it in Chap. 11, *Dense Vector Representations*.

---

<sup>2</sup> <https://github.com/google/cld3>.

# Chapter 8

## Neural Networks



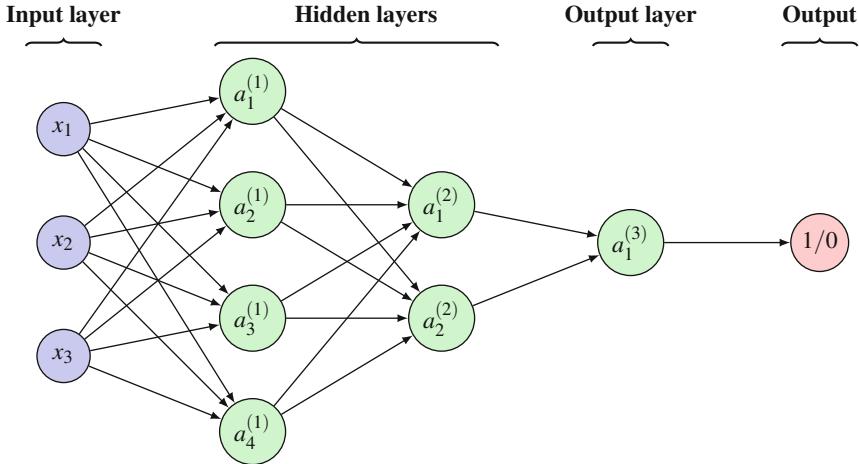
Neural networks form a last family of numerical classifiers that, compared to the other techniques we have seen, have a more flexible and extendible architecture. Typically, neural networks are composed of layers, where each layer contains a set of nodes, the neurons. The input layer corresponds to the 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 all the nodes of the preceding layer, processes this information, and passes it through to all the nodes of the next layer; see Fig. 8.1.

Over the years, neural networks have become one the most efficient machine-learning devices. In this chapter, 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.

### 8.1 Representation and Notation

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.



**Fig. 8.1** Overview of a neural network

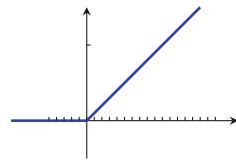
It is easier nonetheless to represent a network with matrices and the following mathematical notation:

- As in the previous sections, let us denote  $\mathbf{x}$  the input vector representing one observation;
- Let us denote  $a_i^{(j)}$  a node at layer  $(j)$ . In Fig 8.1, for instance, the first hidden layer consists of four nodes:  $a_1^{(1)}$ ,  $a_2^{(1)}$ ,  $a_3^{(1)}$ , and  $a_4^{(1)}$ ; The nodes of a hidden layer form a vector that we denote  $\mathbf{a}^{(j)}$ , for instance  $\mathbf{a}^{(1)}$ ;
- Let us call  $\mathbf{w}_i^{(j)}$  the vector representing the weights from the incoming synapses at node  $a_i^{(j)}$ . For instance,  $a_2^{(1)}$  has three incoming connections with weights represented by  $\mathbf{w}_2^{(1)}$ ;
- Given a layer  $(j)$ , we can store all the incoming weights in a matrix that we call  $W^{(j)}$ . This matrix has as many rows as there are hidden nodes in the layer. In the case of Fig. 8.1, we stack the  $\mathbf{w}_1^{(1)}$ ,  $\mathbf{w}_2^{(1)}$ ,  $\mathbf{w}_3^{(1)}$ , and  $\mathbf{w}_4^{(1)}$  vectors as rows of the matrix;
- The size of these matrices will be  $4 \times 3$  for the first hidden layer,  $W^{(1)}$ ,  $2 \times 4$  for the second one,  $W^{(2)}$ , and finally,  $1 \times 2$  for the third layer,  $W^{(3)}$ .

## 8.2 Feed-Forward Computation

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. 8.1, at node

**Fig. 8.2** The reLU function

$a_1^{(1)}$ , the linear combination is:  $\mathbf{w}_1^{(1)} \cdot \mathbf{x}$ . We compute the input values of all the nodes  $a_i^{(1)}$  by multiplying the matrix  $W^{(1)}$  by  $\mathbf{x}$ :

$$W^{(1)}\mathbf{x}.$$

In most networks, we also have an intercept or bias. We implement it with a vector that we add to the product:

$$W^{(1)}\mathbf{x} + \mathbf{b}^{(1)},$$

This bias is not represented in Fig. 8.1;

2. Pass the resulting numbers, a vector, to an **activation function**. This produces the output of the neurons. For the first layer, we have:

$$\text{activation}^{(1)}(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}),$$

resulting in the  $\mathbf{a}^{(1)}$  vector;

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. 8.2), where:

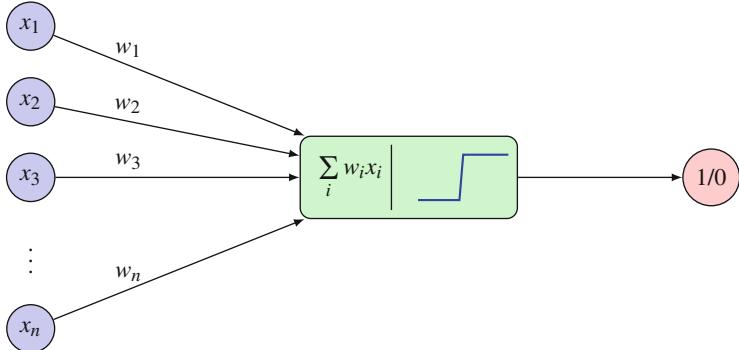
$$\text{reLU}(x) = \max(0, x).$$

We apply the activation function to all the coordinates of the vector. We also say that we map the function over the vector.

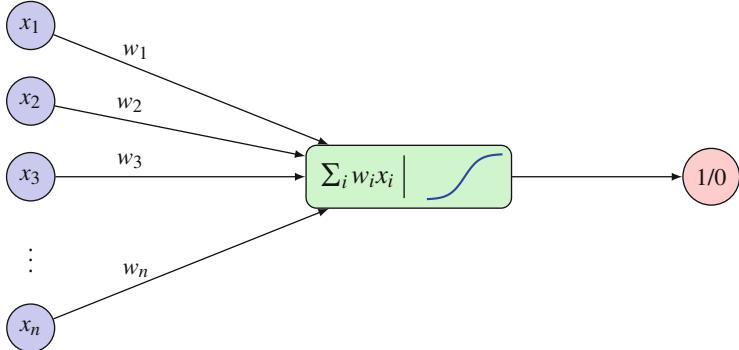
3. For the  $i$ th hidden layer, we apply the same computation as between the input and the first layer and we have the relation:

$$\text{activation}^{(i)}(W^{(i)}\mathbf{a}^{(i-1)} + \mathbf{b}^{(i)}) = \mathbf{a}^{(i)}.$$

The weights matrices  $W^{(j)}$  and bias vectors  $\mathbf{b}^{(j)}$  making the network consist of trainable parameters that we fit using a dataset; see Sect. 8.2.2.



**Fig. 8.3** Neural network representation of the perceptron



**Fig. 8.4** Neural network representation of logistic regression

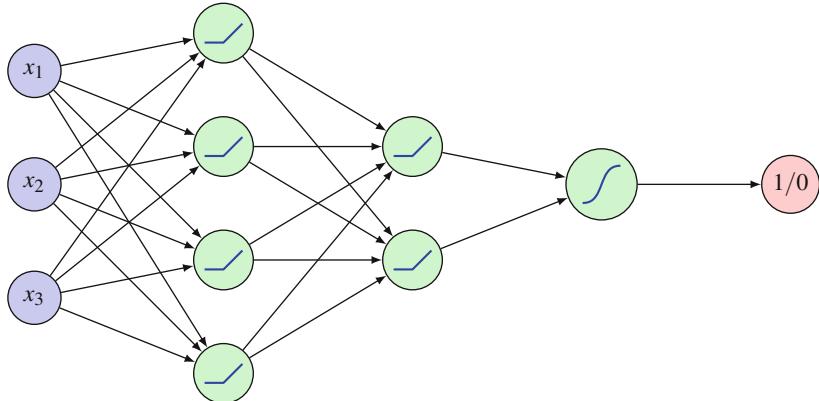
### 8.2.1 *The Perceptron and Logistic Regression*

The perceptron from Sect. 7.8 is the simplest form of neural networks. It has a single layer, where the activation is the Heaviside function. Figure 8.3 shows a graphical representation of it.

We can also reformulate logistic regression from Sect. 7.9 as a neural network. It also has a single layer, where the activation function is the logistic function; see Fig. 8.4.

### 8.2.2 *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



**Fig. 8.5** Network with hidden layers and two activation functions: ReLU and logistic regression

design is to use the ReLU function in the hidden layers, except the last one, where we use the logistic function; see Fig. 8.5.

Training a neural network model is then identical to what we have seen for the perceptron or logistic regression. Given a network topology and activation functions, we apply a training algorithm to a dataset that finds (fits) optimal synaptic weights, i.e. the weight matrices and bias vectors of the network:  $W^{(j)}$  and  $\mathbf{b}^{(j)}$ . To carry this out, the training algorithm uses **backpropagation**: A technique that computes the error gradient iteratively, starting from the last layer, and backpropagates it, one layer at a time, until it reaches the first layer. The update rule from Sect. 7.5, will then enable us to adjust the weights and biases.

## 8.3 Backpropagation

### 8.3.1 Presentation

Backpropagation is a relatively complex mathematical topic. In this book, we will only give an outline of it. We will use Fig. 8.5 along with the text to exemplify it, where we will ignore the biases. In a feedforward network, we have for the first layer:

$$f^{(1)}(W^{(1)}\mathbf{x}),$$

where  $f^{(1)}$  is the activation function at layer 1. For the second layer:

$$f^{(2)}(W^{(2)}f^{(1)}(W^{(1)}\mathbf{x})),$$

until we reach layer  $L$ , the last layer, and output the prediction:

$$\hat{y} = f^{(L)}(W^{(L)} \dots f^{(2)}(W^{(2)} f^{(1)}(W^{(1)} \mathbf{x})) \dots).$$

For the example in Fig. 8.5 and using two linear activations, i.e. no activation, and a final logistic function, we have:

$$\hat{y} = f^{(3)}(W^{(3)} W^{(2)} W^{(1)} \mathbf{x}),$$

where  $f^{(3)}(x)$  is the logistic function.

As with linear regression, we will measure the quality of our network with a loss function, where we will try to minimize the difference between the predicted and observed annotations:  $\text{Loss}(\hat{y}, y)$ .

### 8.3.2 Naive Gradient Descent

As in Sect. 7.5, to fit the matrices, we can compute the partial derivatives with respect to all the weights, i.e. the gradient and apply the gradient descent algorithm until we find a minimal loss. Computing the products of all the network matrices and flattening the  $w_{ij}^{(l)}$  coefficients into one single vector  $\mathbf{w}$ , we would have the following recurrence relation at step  $k$  of the descent:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla \text{Loss}(\mathbf{w}_k).$$

This relation gives us a weight update rule that would enable us to fit the network. Nonetheless, although theoretically possible, it would be difficult to apply this technique in practice as modern neural networks have sometimes billions of parameters.

Coming back to our example in Fig. 8.5 and using the logistic loss, i.e. the binary cross-entropy (Sect. 7.9), the gradient coordinates would be:

$$\begin{aligned} \frac{\partial \text{Loss}(\mathbf{w})}{\partial w_{ij}^{(l)}} &= \frac{\partial (-y \ln \hat{y} - (1-y) \ln(1-\hat{y}))}{\partial w_{ij}^{(l)}} \\ &= \frac{\partial (-y \ln f^{(3)}(W^{(3)} W^{(2)} W^{(1)} \mathbf{x}) - (1-y) \ln(1-f^{(3)}(W^{(3)} W^{(2)} W^{(1)} \mathbf{x})))}{\partial w_{ij}^{(l)}}, \end{aligned}$$

for all the weights  $w_{ij}^{(l)}$ . The complete computation of this expression is left as an exercise to the reader. Again this would yield an update rule to fit our weight parameters, even if this method is impractical in real cases.

### 8.3.3 Breaking Down the Computation

Instead of computing the gradient for the whole network in one shot, we will decompose the problem and compute it one layer at a time. For this, we will follow Le Cun (1987) and proceed in two steps:

1. We will first consider the input and the values in the hidden layers, before and after activation, respectively  $\mathbf{z}^{(l)}$  and  $\mathbf{a}^{(l)}$ , with the equality  $\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)})$ . It is easier to start with the gradient with respect to these input and hidden nodes as the number of parameters in  $\mathbf{a}^{(l)}$  is much lower than that of the weights in  $W^{(l)}$ .
2. Then, we will see that these gradients can serve as intermediate steps to compute the gradients with respect to the weights.

For convenience, we will denote  $\mathbf{a}^{(L)}$  the variable  $\hat{y}$  to have an easier iteration. We have:

$$\begin{aligned}\hat{y} &= \mathbf{a}^{(L)}, \\ &= f^{(L)}(\mathbf{z}^{(L)}), \\ &= f^{(L)}(W^{(L)}\mathbf{a}^{(L-1)})\end{aligned}$$

At this point, it is important to make a distinction between row and column vectors (Sect. 5.5.8). Here, we will represent both the input vector  $\mathbf{x}$  and the values in the hidden layers,  $\mathbf{z}^{(l)}$  and  $\mathbf{a}^{(l)}$ , as column vectors:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{I^{(l)}}^{(l)} \end{bmatrix}, \quad \text{and} \quad \mathbf{a}^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{I^{(l)}}^{(l)} \end{bmatrix}.$$

Their transpose will then be row vectors:

$$\begin{aligned}\mathbf{x}^\top &= [x_1, x_2, \dots, x_n], \\ \mathbf{z}^{(l)\top} &= [z_1^{(l)}, z_2^{(l)}, \dots, z_{I^{(l)}}^{(l)}], \quad \text{and} \\ \mathbf{a}^{(l)\top} &= [a_1^{(l)}, a_2^{(l)}, \dots, a_{I^{(l)}}^{(l)}].\end{aligned}$$

Remember though that row and column vectors are matrices (see Sect. 5.5.8).

### 8.3.4 Gradient with Respect to the Input

As we said, we will proceed backward starting from the output and moving to the input. Let us first consider the last hidden layer and compute the partial derivatives

of the loss,  $\text{Loss}(\hat{y}, y)$ , with respect to  $z_i^{(L)}$ . Using the chain rule, we have:

$$\begin{aligned}\frac{\partial \text{Loss}(\mathbf{a}^{(L)}, y)}{\partial z_i^{(L)}} &= \frac{\partial \text{Loss}(\mathbf{a}^{(L)}, y)}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L)}}{\partial z_i^{(L)}}, \\ &= \frac{\partial \text{Loss}(\mathbf{a}^{(L)}, y)}{\partial \mathbf{a}^{(L)}} f^{(L)'}(z_i^{(L)}).\end{aligned}$$

Computing all the partial derivatives, we obtain the gradient of the loss with respect to  $\mathbf{z}^{(L)}$ :

$$\begin{aligned}\nabla_{\mathbf{z}^{(L)}} \text{Loss}(\mathbf{a}^{(L)}, y) &= \frac{\partial \text{Loss}(\mathbf{a}^{(L)}, y)}{\partial \mathbf{z}^{(L)}}, \\ &= \frac{\partial \text{Loss}(\mathbf{a}^{(L)}, y)}{\partial \mathbf{a}^{(L)}} \odot f^{(L)'}(\mathbf{z}^{(L)}), \\ &= \nabla_{\mathbf{a}^{(L)}} \text{Loss}(\mathbf{a}^{(L)}, y) \odot f^{(L)'}(\mathbf{z}^{(L)}).\end{aligned}$$

where  $\odot$  is the element-wise product, also called Hadamard product.

Going one step backward, we link the two previous hidden layers with the equality:

$$\mathbf{z}^{(L)} = W^{(L)} f^{(L-1)}(\mathbf{z}^{(L-1)}),$$

where each coordinate  $z_i^{(L)}$  of  $\mathbf{z}^{(L)}$  is a scalar function for which we can compute a gradient with respect to  $\mathbf{z}^{(L-1)}$ :

$$\nabla_{\mathbf{z}^{(L-1)}} z_i^{(L)} = \frac{\partial z_i^{(L)}}{\partial \mathbf{z}^{(L-1)}}.$$

In total, we have  $I^{(L)}$  gradients if  $I^{(L)}$  is the number of hidden nodes at layer  $L$ . These gradients are vectors that we can arrange in a matrix called a *Jacobian*. By convention, the gradients will correspond to the rows of the matrix and each cell  $m_{ij}$

will have the value  $\frac{\partial z_i^{(L)}}{\partial z_j^{(L-1)}}$ :

$$J_{\mathbf{z}^{(L-1)}}(\mathbf{z}^{(L)}) = \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} = \begin{bmatrix} \nabla_{\mathbf{z}^{(L-1)}} z_1^{(L)} \\ \nabla_{\mathbf{z}^{(L-1)}} z_2^{(L)} \\ \dots \\ \nabla_{\mathbf{z}^{(L-1)}} z_{I^{(L)}}^{(L)} \end{bmatrix} = \begin{bmatrix} \frac{\partial z_1^{(L)}}{\partial \mathbf{z}^{(L-1)}} \\ \frac{\partial z_2^{(L)}}{\partial \mathbf{z}^{(L-1)}} \\ \dots \\ \frac{\partial z_{I^{(L)}}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial z_1^{(L)}}{\partial z_1^{(L-1)}} & \frac{\partial z_1^{(L)}}{\partial z_2^{(L-1)}} & \cdots & \frac{\partial z_1^{(L)}}{\partial z_{I(L-1)}^{(L-1)}} \\ \frac{\partial z_2^{(L)}}{\partial z_1^{(L-1)}} & \frac{\partial z_2^{(L)}}{\partial z_2^{(L-1)}} & \cdots & \frac{\partial z_2^{(L)}}{\partial z_{I(L-1)}^{(L-1)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{I(L)}^{(L)}}{\partial z_1^{(L-1)}} & \frac{\partial z_{I(L)}^{(L)}}{\partial z_2^{(L-1)}} & \cdots & \frac{\partial z_{I(L)}^{(L)}}{\partial z_{I(L-1)}^{(L-1)}} \end{bmatrix}.$$

Let us now compute the derivatives of the  $\mathbf{z}$  vector at layer  $L$  with respect to a variable in the previous layer:  $z_j^{L-1}$ . It corresponds to the values in the column at index  $j$ :

$$\begin{aligned} \frac{\partial \mathbf{z}^{(L)}}{\partial z_j^{(L-1)}} &= \frac{\partial W^{(L)} f^{(L-1)'}(\mathbf{z}^{(L-1)})}{\partial z_j^{(L-1)}}, \\ &= f^{(L-1)'}(z_j^{(L-1)}) \begin{bmatrix} w_{1,j}^{(L)} \\ w_{2,j}^{(L)} \\ \vdots \\ w_{I^{(L)},j}^{(L)} \end{bmatrix}. \end{aligned}$$

Considering all the columns and computing the partial derivatives with respect to all the  $z_j^{(L-1)}$  variables, we can build the complete Jacobian:

$$\nabla_{\mathbf{z}^{(L-1)}} \mathbf{z}^{(L)} = f^{(L-1)'}(\mathbf{z}^{(L-1)})^\top \odot W^{(L)}.$$

With the Hadamard product, we multiply each element of  $f^{(L-1)'}(\mathbf{z}^{(L-1)})^\top$ , a row vector, with the column of  $W^{(L)}$  of respective index.

We can show that this relation applies for any pair of adjacent layers  $l$  and  $l - 1$  in the network:

$$\nabla_{\mathbf{z}^{(l-1)}} \mathbf{z}^{(l)} = f^{(l-1)'}(\mathbf{z}^{(l-1)})^\top \odot W^{(l)}.$$

We can now use the chain rule for composed functions to compute  $\nabla_{\mathbf{x}} Loss(\mathbf{y}, \hat{\mathbf{y}})$  as the product of the intermediate gradients with respect to the hidden cells:

$$\begin{aligned} \nabla_{\mathbf{x}} Loss(\hat{\mathbf{y}}, \mathbf{y}) &= \nabla_{\mathbf{x}} Loss(f^{(L)}(W^{(L)} \dots f^{(2)}(W^{(2)} f^{(1)}(W^{(1)} \mathbf{x})) \dots), \mathbf{y}), \\ &= \frac{\partial Loss(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \frac{\partial \mathbf{z}^{(L-1)}}{\partial \mathbf{z}^{(L-2)}} \cdots \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{x}}, \\ &= \nabla_{\mathbf{a}^{(L)}} Loss(\mathbf{a}^{(L)}, \mathbf{y}) \odot f^{(L)'}(\mathbf{z}^{(L)}) f^{(L-1)'}(\mathbf{z}^{(L-1)})^\top \\ &\quad \odot W^{(L)} \dots f^{(1)'}(\mathbf{z}^{(1)})^\top \odot W^{(2)} W^{(1)}. \end{aligned}$$

More generally, for a hidden layer at index  $l$ , we have:

$$\begin{aligned}\nabla_{\mathbf{z}^{(l)}} \text{Loss}(\hat{y}, y) &= \nabla_{\mathbf{z}^{(l)}} \text{Loss}(f^{(L)}(W^{(L)} \dots f^{(l+2)}(W^{(l+2)} f^{(l+1)} \\ &\quad \times (W^{(l+1)} \mathbf{z}^{(l)})) \dots), y), \\ &= \nabla_{\mathbf{a}^{(L)}} \text{Loss}(\mathbf{a}^{(L)}, y) \odot f^{(L)\top}(\mathbf{z}^{(L)}) f^{(L-1)\top}(\mathbf{z}^{(L-1)})^\top \\ &\quad \odot W^{(L)} \dots f^{(l)\top}(\mathbf{z}^{(l)})^\top \odot W^{(l+1)}, \\ &= \nabla_{\mathbf{z}^{(l+1)}} \text{Loss}(\hat{y}, y) f^{(l)\top}(\mathbf{z}^{(l)})^\top \odot W^{(l+1)}.\end{aligned}$$

This last term shows that we can recursively compute, **backpropagate**, the gradient with respect to the hidden values from the output until we reach the input. Note that  $\nabla_{\mathbf{z}^{(l)}} \text{Loss}(\hat{y}, y)$  is sometimes denoted  $\delta^{(l)}$ .

Using our example in Fig. 8.5, two linear activations instead of ReLU,  $f^{(1)}$  and  $f^{(2)}$ , a final logistic function,  $f^{(3)}$ , and a logistic loss, we have  $f^{(1)\top} = f^{(2)\top} = 1$ ,  $f^{(3)\top} = f^{(3)}(1 - f^{(3)})$ , and  $\frac{dL(\hat{y}, y)}{d\hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$  (see Sect. 7.9.1).

The gradient of the loss with respect to  $\mathbf{x}$  is :

$$\begin{aligned}\nabla_{\mathbf{x}} \text{Loss}(\hat{y}, y) &= -\frac{\partial(y \ln \hat{y} + (1-y) \ln(1-\hat{y}))}{\partial \mathbf{x}}, \\ &= \nabla_{\hat{y}} \text{Loss}(\hat{y}, y) \hat{y}' W^{(3)} W^{(2)} W^{(1)}, \\ &= \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y}) W^{(3)} W^{(2)} W^{(1)}, \\ &= (\hat{y} - y) W^{(3)} W^{(2)} W^{(1)}, \\ &= (f^{(3)}(W^{(3)} W^{(2)} W^{(1)} \mathbf{x}) - y) W^{(3)} W^{(2)} W^{(1)}.\end{aligned}$$

### 8.3.5 Gradient with Respect to the Weights

Now that we have the gradient with respect to the input and hidden layers, let us define the backpropagation algorithm, where we compute the gradient with respect to  $W^{(l)}$ ,  $l$  being the index of any layer. From the chain rule, for the last layer,  $L$ , we have:

$$\nabla_{W^{(L)}} \text{Loss}(\hat{y}, y) = \frac{\partial \text{Loss}(\hat{y}, y)}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial W^{(L)}}$$

and

$$\begin{aligned}\mathbf{z}^{(L)} &= W^{(L)} f^{(L-1)}(\mathbf{z}^{(L-1)}), \\ &= W^{(L)} \mathbf{a}^{(L-1)}.\end{aligned}$$

The partial derivatives of  $\mathbf{z}^{(L)}$  with respect to  $W^{(L)}$  simply consist of the transpose of  $\mathbf{a}^{(L-1)}$ . Then, we have:

$$\frac{\partial \mathbf{z}^{(L)}}{\partial W^{(L)}} = \mathbf{a}^{(L-1)\top}.$$

We can now compute the gradient of the loss with respect of  $W^{(L)}$ :

$$\nabla_{W^{(L)}} Loss(\hat{y}, y) = \nabla_{\mathbf{a}^{(L)}} Loss(\mathbf{a}^{(L)}, y) f^{(L)'}(\mathbf{z}^{(L)}) \mathbf{a}^{(L-1)\top},$$

which enables us to apply the update rule to all the weights of the last layer using the  $\mathbf{a}^{(L-1)}$ ,  $\mathbf{a}^{(L)}$ , and  $\mathbf{z}^{(L)}$  values we have stored in the forward pass.

With our example in Fig. 8.5, this corresponds to:

$$\begin{aligned}\nabla_{W^{(3)}} Loss(\hat{y}, y) &= -\frac{\partial(y \ln \hat{y} + (1-y) \ln(1-\hat{y}))}{\partial W^{(3)}}, \\ &= \nabla_{\mathbf{a}^{(3)}} Loss(\mathbf{a}^{(3)}, y) f^{(3)'}(\mathbf{z}^{(3)}) \mathbf{a}^{(2)\top}, \\ &= (\hat{y} - y) \mathbf{a}^{(2)\top}, \\ &= (f^{(3)}(W^{(3)} W^{(2)} W^{(1)} \mathbf{x}) - y)(W^{(2)} W^{(1)} \mathbf{x})^\top.\end{aligned}$$

Coming back to the general case, we can now proceed backward using the chain rule for any  $l$ :

$$\nabla_{W^{(l)}} Loss(\hat{y}, y) = \frac{\partial Loss(\hat{y}, y)}{\partial \mathbf{z}^{(l)}} \frac{\partial \mathbf{z}^{(l)}}{\partial W^{(l)}}$$

with

$$\mathbf{z}^{(l)} = W^{(l)} \mathbf{a}^{(l-1)}$$

and

$$\frac{\partial \mathbf{z}^{(l)}}{\partial W^{(l)}} = \mathbf{a}^{(l-1)\top}.$$

Finally, we have

$$\nabla_{W^{(l)}} Loss(\hat{y}, y) = \nabla_{\mathbf{z}^{(l)}} Loss(\hat{y}, y) \mathbf{a}^{(l-1)\top}.$$

that gives us the value of the gradient of the weights at index  $l$ . This gradient enables us to apply the update rule to the weight matrices.

## 8.4 Applying Neural Networks to Datasets

In practical cases, we will apply networks to datasets or to minibatches. As we have seen in Sects. 7.4 and 7.11.1, datasets are arranged by rows, where each row is an observation. In our description of feed-forward computation, in Sect. 8.2, we represented the  $\mathbf{x}$  input as a column vector (a one-column matrix).

It is easy to convert the input so that it fits the dataset format. We just have to transpose the matrix product:

$$(W\mathbf{x})^\top = \mathbf{x}^\top W^\top, \\ = \hat{y}.$$

$\mathbf{x}^\top$  is now a one-row matrix and we can apply the product to the whole  $X$  dataset by stacking all the input rows vertically. This product yields the predicted outputs:

$$XW^\top = \hat{y}.$$

Extending this to a network of  $L$  layers, and ignoring the biases, we have:

$$\begin{aligned} & f^{(L)}(W^{(L)} \dots f^{(2)}(W^{(2)}f^{(1)}(W^{(1)}\mathbf{x})) \dots)^\top \\ &= f^{(L)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}^\top W^{(1)\top})W^{(2)\top}) \dots W^{(L)\top}), \\ &= \hat{y}, \end{aligned}$$

and for the whole  $X$  dataset:

$$f^{(L)}(\dots f^{(2)}(f^{(1)}(XW^{(1)\top})W^{(2)\top}) \dots W^{(L)\top}) = \hat{y}.$$

For the example in Fig. 8.5, this yields:

$$f^{(3)}(XW^{(1)\top}W^{(2)\top}W^{(3)\top}) = \hat{y},$$

where  $f^{(3)}(x)$  is the logistic function.

## 8.5 Programming Neural Networks

We will now apply the concepts we have learned to the *Salammbô* dataset. For this programming part, we will start with Keras (Chollet 2021), as it is a smooth transition with scikit-learn, and then we will use PyTorch (Paszke et al. 2019) for the rest of this book. Both are comprehensive libraries of neural network algorithms. Keras is in fact a high-level programming interface on top of either Tensorflow (Abadi et al. 2016), PyTorch, or JAX.

Before we fit a model, we will normalize the data. This part is common to Keras and PyTorch. We will then build identical networks with both libraries so that we can compare their programming interfaces.

### 8.5.1 Data Representation and Preprocessing

We store the *Salammbô* dataset,  $X$  and  $y$ , in NumPy arrays using the same structure as in Sect. 7.11.1. The algorithms in Keras and PyTorch are quite sensitive to differences in numeric ranges between the features. Prior to fitting a model, a common practice is to standardize the columns, here 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 of the rows, i.e. divide the chapter vectors by their norm, before the standardization. As a result, all the chapter rows will 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 will rely on scikit-learn and its built-in classes, `Normalizer` and `StandardScaler`, to normalize and standardize an array. These classes have two main methods: `fit()` and `transform()`. We first use `fit()` to determine the parameters of the normalization or standardization and then `transform()` to apply them to the dataset. We normally use `fit()` once and `transform()` as many times as we need to standardize the data.

We can combine `fit()` and `transform()` in the `fit_transform()` sequence:

```
from sklearn.preprocessing import StandardScaler, Normalizer

X_norm = Normalizer().fit_transform(X)
X_scaled = StandardScaler().fit_transform(X_norm)
```

### 8.5.2 Keras

#### Building the Network

Building a feed-forward network is easy with Keras. We just need to describe its structure in terms of layers. The fully connected nodes correspond to `Dense` layers that we will assemble with the `Sequential` class. To implement logistic regression, we then use a `Sequential` model with one `Dense()` layer in the sequence. In the `Dense()` object, we specify the output dimension as first parameter, 1, for English (0) or French (1), then, using a named argument, 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:

```
import numpy as np
import keras_core as keras

np.random.seed(0)
model = keras.Sequential([
    keras.layers.Dense(1, activation='sigmoid')
])
```

#### Setting the Gradient Descent Parameters

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 `sgd` for stochastic gradient descent. The fitting process will report the loss for each epoch. We also tell it to report the accuracy with the `metrics` argument.

```
model.compile(loss='binary_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

The optimizer has often a big influence on the results of the descent. Keras supports the most common algorithms such as RMSprop, `rmsprop`, or Adam, `adam`, that we described in Sect. 7.10. Here, they would produce the same results as `sgd`.

#### Training the Model

Once the model is compiled, we train it with the `fit()` method, where we pass the scaled dataset, the number of epochs, and the batch size. For a batch gradient descent, `batch_size` would be set to the size of the dataset. Here we apply a stochastic descent and we set `batch_size` to 1. In most applications, we would use

minibatches (Sect. 7.5), where the batch size should have higher values, such as 8, 16, 32, or 64.

```
history = model.fit(X_scaled, y, epochs=20, batch_size=1)
```

The returned variable `history.history` is a dictionary with two keys: the loss and the accuracy. We plot the loss with these statements:

```
import matplotlib.pyplot as plt

plt.scatter(range(len(history.history['loss'])),
            history.history['loss'], c='b', marker='x')
plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('BCE')
plt.show()
```

and we just change `loss` with `accuracy` to plot the accuracy (Fig. 8.6).

We can see that with such a small dataset, the loss decreases steadily and the accuracy reaches one immediately. A more realistic experiment would use validation data. We would include them in the fitting code with these arguments:

```
history = model.fit(X_train, y_train,
                     epochs=20,
                     batch_size=1,
                     validation_data=(X_val, y_val))
```

and we would plot `val_loss` and `val_accuracy`.

## Predicting Classes

Once trained, we can apply the model to a dataset with the `predict()` method, here to the training set again.

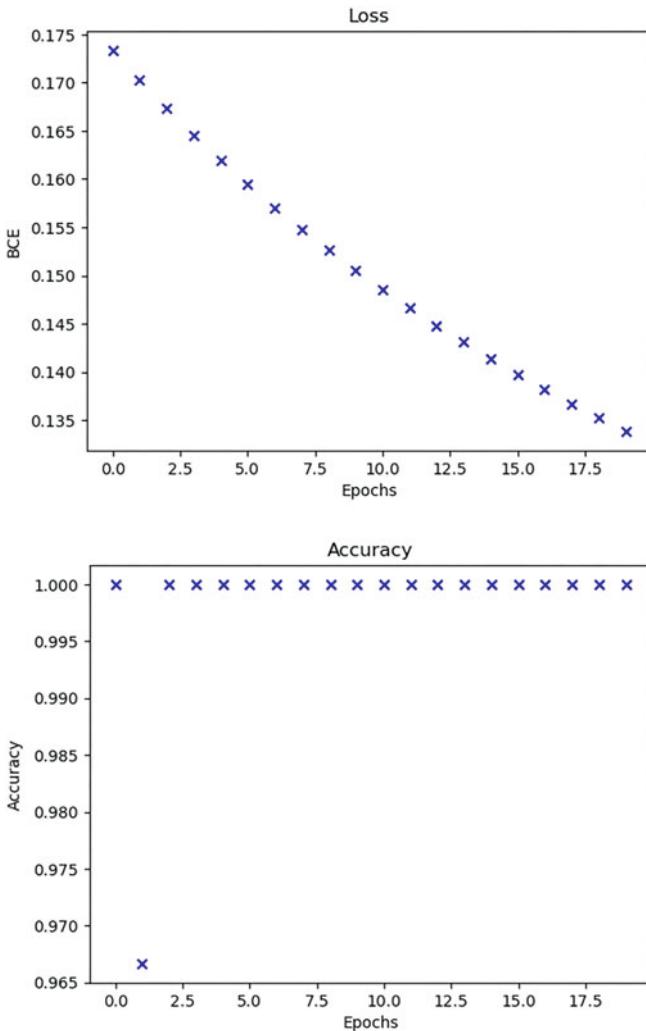
```
y_pred_proba = model.predict(X_scaled)
```

which returns the probability of class 1:

```
array([[0.00461974],
       [0.14686179],
       [0.21391678],
       ...
       [0.98046577],
       [0.5141997 ]], dtype=float32)
```

We predict class 1 when the probability is greater than a threshold of 0.5 and 0, when it is less:

```
def predict_class(y_pred_proba):
    y_pred = np.zeros(y_pred_proba.shape[0])
    for i in range(y_pred_proba.shape[0]):
        if y_pred_proba[i][0] >= 0.5:
            y_pred[i] = 1
    return y_pred
```



**Fig. 8.6** The training loss and accuracy over the epochs

In this small example, where we train and apply a model on the same dataset, we reach an accuracy of 100%.

```
predict_class(y_pred_proba)
# array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...
#        ... 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

We can also evaluate the model using Keras `evaluate()` and report the binary cross-entropy and the accuracy:

```
# evaluate the model
scores = model.evaluate(X_scaled, y)
# [0.13300223648548126, 1.0]
```

Remember that if you run this program on your machine, as the initialization is random, the figures will be different.

## Model Parameters

The model parameters consist the matrix weights, here a column vector, and the bias that we obtain with `model.get_weights()`. As the input has two parameters, the number of letters and the number of As, our vector has two coordinates. The bias has only one coordinate:

```
>>> model.get_weights()

[array([[-0.7931856],
       [ 1.1318369]], dtype=float32),
 array([0.00644173], dtype=float32)]
```

## Adding Hidden Layers

So far, we used a single layer. If we want to add hidden layers, for instance to insert a layer of five nodes, we just declare them in a list that we pass to the `Sequential` class. As mentioned in Sect. 8.2.2, intermediate layers should use the `relu` activation function. We do not need to specify the input size of intermediate layers; Keras automatically infers it:

```
model = keras.Sequential([
    layers.Dense(5, activation='relu'),
    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.

### 8.5.3 PyTorch

After scikit-learn and Keras, PyTorch is third machine-learning library. Like Keras, it is intended to deep-learning applications but their application programming

interfaces (APIs) are quite different. With PyTorch, the programmer has to manage lower level details, that makes it more flexible, but also more difficult to program.

PyTorch does not use NumPy arrays. Instead, it has an equivalent data structure called tensors that we saw in Chap. 5, *Python for Numerical Computations*. As they are not identical, we need to convert our dataset in this format. In addition, because of loss output format of binary cross-entropy in PyTorch, the  $y$  vector must be a column vector:

```
import numpy as np
import torch

Y = y.reshape((-1, 1))

X_scaled = torch.from_numpy(X_scaled).float()
Y = torch.from_numpy(Y).float()
```

## Building the Network with Sequential

Similarly to Keras, PyTorch has a `Sequential` class, where we declare a processing pipeline. Our simplest model, logistic regression, has only one layer with two input parameters and one output:

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(2, 1)
)
```

`Linear` is a fully connected layer, where the arguments are the input and output dimensions. A `Linear` object consists of parameters representing a  $W$  matrix and a  $\mathbf{b}$  bias. In Sect. 5.8, we saw the application of this matrix to an input  $X$  corresponds to:  $XW^T + \mathbf{b}$ . A subsequent fitting procedure will determine the parameters of  $W$  and  $\mathbf{b}$ .

## Setting the Gradient Descent Parameters

Once we have defined the model architecture, we specify the parameters of the gradient descent: The loss to optimize and the optimization algorithm. As with Keras, this is the binary cross-entropy and the stochastic gradient descent.

We first create the loss function with the statement;

```
loss_fn = nn.BCEWithLogitsLoss()      # binary cross entropy loss
```

It incorporates the logistic function and this explains its long name here: `BCEWithLogitsLoss()`. As input, this loss function uses the output of the linear layer of `model`. Such an output is called a logit, which is the inverse of the logistic function. Although this term is a bit convoluted, it is standard in the field. As an

analogy, we could think of the sine function and call the input arcsine instead of angle.

A true binary cross-entropy loss, as we defined it in Sect. 7.9.1, also exists in PyTorch under the name `BCELoss()`, but it is not recommended. The reason is that a naive computation of the logistic function  $\frac{1}{1+e^{-x}}$  is unstable even with relatively small numbers like  $-1000$  for which `math.exp(1000)` throws an overflow error. The cross-entropy from logits uses a technique called the log-sum-exp trick that factors terms and makes the computation possible.

We then select an optimizer, here stochastic gradient descent, where we give the parameters to optimize `model.parameters()`, corresponding here to  $W$  and  $\mathbf{b}$ , and the learning rate, `lr`:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

## Compiling the Model

In version 2.0, PyTorch introduced an optional compiling of the model code that should make its execution faster. If we want to compile your model, we just add one statement before you train it:

```
model = torch.compile(model)
```

Note however that the speedup will depend on the underlying execution platform and this is still a new feature that may be unstable.

## Training the Model

Now that we have a model, a loss, and an optimizer, we can fit the parameters. As opposed to scikit-learn and Keras, we have to write a loop to iterate explicitly over the epochs of the gradient descent. For a batch descent, we have the sequence:

```
model.train()                      # sets PyTorch in the train mode
bce_loss = []
for epoch in range(30):
    Y_pred = model(X_scaled)
    loss = loss_fn(Y_pred, Y)
    bce_loss += [loss.item()]
    optimizer.zero_grad()      # resets the gradients
    loss.backward()            # gradient backpropagation
    optimizer.step()          # weight updates
```

where we first set the model in the training mode with `model.train()`, then `model()` applies the forward pass to the input and `loss_fn()` computes the loss between the predicted and true values. The two next statements backpropagate the gradient as in Sect. 8.3. By default, PyTorch accumulates the gradients and `zero_grad()` resets their values to zero. Finally, `step()` updates the weights as in Sect. 7.5. We record the loss evolution so that we can plot it as in Fig. 8.6.

## Predicting Classes

Once trained, we can apply the model to an input, here the training set `X_scaled`, to predict classes. This is called the inference or evaluation mode. Before we start the prediction, we tell PyTorch we are in this mode with `model.eval()`. This will skip some operations carried out in the training step. Then, we compute the matrix product and output the logits:

```
model.eval()
y_pred_logits = model(X_scaled) # applies the model
```

We need then to apply a logistic function to obtain the probabilities and predict the class:

```
y_pred_proba = torch.sigmoid(y_pred_logits)
```

During inference, it is a common practice to disable the gradient computation as we will not backpropagate it. We use a `with` block with `torch.no_grad()` for this. Finally, the complete evaluation code is:

```
model.eval()
with torch.no_grad():      # disables gradient calculation
    y_pred_logits = model(X_scaled)
    y_pred_proba = torch.sigmoid(y_pred_logits)
y_pred_proba
```

As with Keras, this returns a tensor of probabilities to belong to class 1:

```
tensor([[0.2562],
       [0.4444],
       ...
       [0.7542],
       [0.5508]])
```

We apply a threshold of 0.5 to obtain the final vector of classes.

## Model Parameters

The model has a structure nearly identical to that of Keras. We obtain it with

```
>>> list(model.parameters())
[Parameter containing:
 tensor([[-0.6147,  0.1383]], requires_grad=True),
 Parameter containing:
 tensor([0.2068], requires_grad=True)]
```

that outputs a generator. We can also print the parameters as an ordered dictionary, a dictionary, where the keys are ordered:

```
>>> model.state_dict()
OrderedDict([('0.weight', tensor([-0.6147,  0.1383])),
             ('0.bias', tensor([0.2068]))])
```

## Adding Hidden Layers

We add hidden layers similarly to Keras, but we have to specify the complete size of the matrices and the activation, here `ReLU()`, in the sequence:

```
model = nn.Sequential(
    nn.Linear(2, 5),
    nn.ReLU(),
    nn.Linear(5, 1)
)
```

We may want to name the layers in the network above. We use this statement then:

```
from collections import OrderedDict

model = nn.Sequential(OrderedDict([
    ('W1', nn.Linear(2, 5)),  # W1x + b1
    ('relu', nn.ReLU()),      # relu(W1x + b1)
    ('W2', nn.Linear(5, 1))   # W2relu(W1x + b1) + b2
]))
```

In Sect. 5.8, we saw that applying this network to the  $X$  dataset corresponds to the matrix product:

$$\text{relu}(XW^{(1)\top} + \mathbf{b}^{(1)})W^{(2)\top} + \mathbf{b}^{(2)}.$$

As with our first PyTorch model, we omit the last logistic activation and we return the logits. The training loop and the prediction are identical to those of our first model.

## PyTorch Dataloaders

We used a loop to iterate over the epochs in the batch descent. If we want to apply a stochastic descent or use minibatches, we have to write an inner loop as this one for a batch size of 4. In this inner loop, the sequence of statements is the same:

```
BATCH_SIZE = 4
model.train()
for epoch in range(50):
    for i in range(0, X_scaled.size(dim=0), BATCH_SIZE):
        Y_batch_pred = model(X_scaled[i:i + BATCH_SIZE])
        loss = loss_fn(Y_batch_pred, Y[i:i + BATCH_SIZE])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

However, experience shows that the convergence is more stable if we shuffle  $X$  and  $y$ . For this, we can use two PyTorch utility classes: `Dataset` and `DataLoader`. The combination of both will enable us to manage large quantities of data and iterate over them. We will here introduce their simplest features.

`TensorDataset` is a subclass of `Dataset` that models perfectly the data structure we have. We create a dataset object with the `X` and `y` tensors as input. `DataLoader` creates an iterable object from the dataset. This object will then supply the training inner loop with shuffled minibatches:

```
from torch.utils.data import TensorDataset, DataLoader

dataset = TensorDataset(X_scaled, Y)
dataloader = DataLoader(dataset,
                       batch_size=BATCH_SIZE,
                       shuffle=True)
```

The inner loop to apply the descent with minibatches is now easier to write:

```
model.train()
for epoch in range(50):
    for X_scaled_batch, Y_batch in dataloader:
        Y_batch_pred = model(X_scaled_batch)
        loss = loss_fn(Y_batch_pred, Y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

## Building the Network: Deriving the Module Class

We created our first PyTorch feed-forward networks with the `Sequential` class. This is the most simple way if the network is just a pipeline of elementary components. This is not always the case and we will see here another kind of implementation, where we will encapsulate and create all the elementary components we need in a dedicated class and tell explicitly how to apply them in the forward pass.

PyTorch uses the `nn.Module` base class for all its neural networks. To create a new model, we subclass it and implement two methods: `__init__()` and `forward()`. For the logistic regression model:

1. In `__init__()`, we create the layers of our network, here one `Linear` layer;
2. The `forward()` method computes the matrix product  $XW^T + \mathbf{b}$  from the `X` input, as described in Sect. 8.4.

```
class Model(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 1)

    def forward(self, x):
        return self.fc1(x)
```

The model so far is just a class. To run it, we create an instance of it, choose a loss function, here binary cross entropy, and select an optimizer. We create the model object with:

```
input_dim = X_scaled.size(dim=1)
model = Model(input_dim)
```

The rest is identical to the `Sequential()` model:

```
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

The fitting procedure is also the same.

## Adding Hidden Layers

To insert a hidden layer of five nodes in our previous model, we simply add a `Linear` layer in the class. In the `forward()` method, we insert a `relu()` after the first layer and we apply the second one:

```
class Model(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 5)
        self.fc2 = nn.Linear(5, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

## 8.6 Classification with More than Two Classes

In Sects. 8.5.2 and 8.5.3, we used the logistic function to carry out a classification with two languages and binary cross entropy as a loss. Now, how can we do if instead of a binary output, French or English, we have three or more languages to detect: Latin, Greek, German, Russian, etc., possibly all the languages in wikipedia?

The answer is that we will need to define two more general functions to quantify the loss and to estimate the probability of a vector  $\mathbf{x}$  to belong to a certain class, here  $i$ ,  $P(y = i|\mathbf{x})$ . This generalization of logistic regression is called **multinomial** or **multiclass logistic regression** and, fortunately, it will not change much the structure of our programs.

### 8.6.1 Cross Entropy Loss

In Sect. 7.9, we saw that binary logistic regression minimizes the binary cross entropy loss. Categorical cross entropy or simply cross entropy is an extension of it that covers multiple classes, where we compute the loss for each class and we sum or average all the losses. The mean is defined as:

$$CELoss(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{q} \sum_{j=1}^q \mathbf{y}_j \cdot \ln \hat{\mathbf{y}}_j,$$

where  $\mathbf{y}_j$  corresponds to the true class of the  $j$ th observation, represented as a one-hot vector (Sect. 6.3),  $\hat{\mathbf{y}}_j$  is the probability distribution of the predicted classes, and  $q$ , the number of observations.

Supposing we have three languages in our dataset, English, French, and German, in this order, we first associate them to an index: 0, 1, 2 and then convert these indices to one-hot vectors.  $\mathbf{y}$  will have three possible values:

1.  $\mathbf{y} = (1, 0, 0)$  for English,
2.  $\mathbf{y} = (0, 1, 0)$  for French, and
3.  $\mathbf{y} = (0, 0, 1)$  for German.

Given an observation  $\mathbf{x}$ , the classifier will output a three-dimensional vector estimating the probabilities to belong to a certain class:

$$\hat{\mathbf{y}} = (P(\mathbf{x}|\text{English}), P(\mathbf{x}|\text{French}), P(\mathbf{x}|\text{German})),$$

for instance  $\hat{\mathbf{y}} = (0.2, 0.7, 0.1)$ .

Let us now compute a loss with a made-up dataset of three observations: A text in English, where we encode the true value as  $\mathbf{y}_1 = (1, 0, 0)$ , in French with  $\mathbf{y}_2 = (0, 1, 0)$ , and in German with  $\mathbf{y}_3 = (0, 0, 1)$ . Let us suppose that our classifier predicts the languages with the respective probability estimates:  $\hat{\mathbf{y}}_1 = (0.5, 0.3, 0.2)$ ,  $\hat{\mathbf{y}}_2 = (0.2, 0.7, 0.1)$ , and  $\hat{\mathbf{y}}_3 = (0.3, 0.5, 0.2)$ . Using the mean, the cross entropy loss of these predictions is:

$$\begin{aligned} CELoss(\hat{\mathbf{y}}, \mathbf{y}) &= -\frac{1}{3}(\mathbf{y}_1 \cdot \ln \hat{\mathbf{y}}_1 + \mathbf{y}_2 \cdot \ln \hat{\mathbf{y}}_2 + \mathbf{y}_3 \cdot \ln \hat{\mathbf{y}}_3), \\ &= -\frac{1}{3}((1, 0, 0) \cdot (\ln 0.5, \ln 0.3, \ln 0.2) + \\ &\quad (0, 1, 0) \cdot (\ln 0.2, \ln 0.7, \ln 0.1) + \\ &\quad (0, 0, 1) \cdot (\ln 0.3, \ln 0.5, \ln 0.2)), \\ &= -\frac{1}{3}(\ln 0.5 + \ln 0.7 + \ln 0.2), \\ &\approx 0.89. \end{aligned}$$

We will see a practical example of computation of cross entropy in Sect. 8.7.

### 8.6.2 The Softmax Function

The predicted probability for  $\mathbf{x}$  to belong to class  $i$ ,  $P(y = i|\mathbf{x})$ , is given by a generalization of the logistic function called the **softmax function**.<sup>1</sup> This softmax function takes a vector  $\mathbf{z} = (z_1, z_2, \dots, z_C)$  as input and returns a normalized vector whose  $i$ th coordinate is:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^C e^{z_j}}.$$

It is easy to check that

$$\sum_{i=1}^C \sigma(\mathbf{z})_i = 1.$$

Multinomial logistic regression corresponds to a network with only one layer. As model parameters, we will have as many weight vectors as we have languages or classes to identify. For  $C$  languages, we will have vectors  $\mathbf{w}_1$  to  $\mathbf{w}_C$ , where  $\mathbf{w}_i$  corresponds to language  $i$ . For an input vector  $\mathbf{x}$ , the softmax probability distribution over the  $C$  classes is:

$$\sigma(\mathbf{w}_1 \cdot \mathbf{x}, \mathbf{w}_2 \cdot \mathbf{x}, \dots, \mathbf{w}_C \cdot \mathbf{x}) = \left( \frac{e^{\mathbf{w}_1 \cdot \mathbf{x}}}{\sum_{j=1}^C e^{\mathbf{w}_j \cdot \mathbf{x}}}, \frac{e^{\mathbf{w}_2 \cdot \mathbf{x}}}{\sum_{j=1}^C e^{\mathbf{w}_j \cdot \mathbf{x}}}, \dots, \frac{e^{\mathbf{w}_C \cdot \mathbf{x}}}{\sum_{j=1}^C e^{\mathbf{w}_j \cdot \mathbf{x}}} \right).$$

The probability for input vector  $\mathbf{x}$  to have  $i$  as language, for instance French or German, is then simply the  $i$ th coordinate of the softmax distribution:

$$P(y = i|\mathbf{x}) = \frac{e^{\mathbf{w}_i \cdot \mathbf{x}}}{\sum_{j=1}^C e^{\mathbf{w}_j \cdot \mathbf{x}}}$$

A training procedure based on gradient descent determines the weight values  $\mathbf{w}_j$  attached to each language  $j$ .

If our network has more than one layer, we apply the softmax function to the last layer. All the previous layers normally use the `relu` activation.

---

<sup>1</sup> The softmax function is in fact a renaming of the much older Boltzmann distribution.

## 8.7 Multiclass Classification with Keras

The Keras functions for multiclass networks are nearly identical to those of binary networks. We just change the loss from `binary_crossentropy` to `categorical_crossentropy` and the activation of the last layer from `sigmoid` to `softmax`. However, Keras does not handle the output labels like scikit-learn and we need to convert them into a Keras-compatible format. There are two options: one-hot vectors or integer indices:

1. We can represent the class of an observation,  $y$ , as a one-hot vector, just as in Sect. 8.6 with, for example,  $y = (1, 0, 0)$  for English. Given an observation  $x$ , the classifier outputs the probabilities to belong to a certain class for instance  $\hat{y} = (0.2, 0.7, 0.1)$ . We predict the class by picking the index with the maximal value,  $\arg \max_i \hat{y}$ , here  $i = 1$ , French, with a probability of 0.7. As the index of the true language is 0 in this example, the loss is of  $-\ln 0.2$ .
2. One-hot vectors are equivalent to indices and we can also encode  $y$  with a number, for instance 0, 1, and 2 for three classes. In this case, we replace the loss argument with `sparse_categorical_crossentropy`.

We will now write a program to exemplify these concepts. We will use the same dataset as with scikit-learn in Sect. 7.11.8 with letter counts of versions of *Salammbô* in French, English, and German. We store them in an  $x$  matrix and, in the  $y$  vector, the category of these counts:

```
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,
     2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Before we fit a model, we scale  $x$  as in Sect. 8.5.1 and we store the result in `X_scaled`. We convert  $y$  in one-hot vectors using the utility function `to_categorical`:

```
Y_cat = keras.utils.to_categorical(y)
```

which returns the matrix:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       ...,
       [0., 0., 1.],
       [0., 0., 1.]])
```

The model architecture is the same as in Sect. 8.5.2 except the arguments of the last layer, where we change the number of output classes to 3 and the activation function from `sigmoid` to `softmax`. For one layer, we have:

```
model = keras.Sequential([
    keras.layers.Dense(3, activation='softmax')
])
```

and with one hidden layer:

```
model = keras.Sequential([
    keras.layers.Dense(5, activation='relu'),
    keras.layers.Dense(3, activation='softmax')
])
```

We can now compile and fit the model. This part is also nearly identical to the code in Sect. 8.5.2: We just need to change the loss to `categorical_crossentropy` as we have more than two classes:

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
model.fit(X_scaled, Y_cat, epochs=50, batch_size=1)
```

Once we have fitted the model, we can predict the class of new observations. In the code below, we reapply the model to the training set:

```
Y_pred_proba = model.predict(X_scaled)
```

The model with one hidden layer returns the matrix:

```
array([[0.692, 0.075, 0.233],
       [0.606, 0.273, 0.121],
       ...
       [0.005, 0.004, 0.991]], dtype=float32)
```

Each row represent the probability estimates of an observation. To extract the class, we pick the index of the highest value. In first row, this is index 0, representing English, with a probability of 0.692. In the last one, this is index 2, German, with a probability of 0.991.

We extract the classes from a matrix of probability estimates with the `np.argmax()` function:

```
y_pred = np.argmax(Y_pred_proba, axis=-1)
```

returning  $\hat{y}$ :

```
np.array(
    [0, 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, 1,
     2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2])
```

Compared with  $y$ ,  $\hat{y}$  contains three errors as the model predicted wrongly three German samples. This yields an accuracy of 93%

Although accuracy is intuitive, the gradient descent only tries to minimize the loss. For one observation,  $(x_i, y_i)$ , where  $y_i$  is the class represented as a one-hot vector, this corresponds to the dot product:

$$-\mathbf{y}_i \cdot \ln \hat{\mathbf{y}}_i$$

as we saw in Sect. 8.6. For the first row, we compute its value with the line:

```
-[1, 0, 0] @ np.log([0.692, 0.075, 0.233]) # 0.368
```

For the whole dataset, we extract the probabilities of the true classes of the observations, compute their logarithms and their mean:

```
-np.mean(np.log(Y_pred_proba[range(0, len(y)), y]))
```

We obtain a loss of 0.32. We can also compute the loss and the accuracy with

```
model.evaluate(X_scaled, Y_cat)
```

## 8.8 Multiclass Classification with PyTorch

The creation and standardization of the dataset is the same as with Keras. We need though to convert the data to PyTorch tensors:

```
X_scaled = torch.from_numpy(X_scaled).float()
y = torch.from_numpy(y).long()
```

We represent the `y` vector as a one-dimensional tensor of long integers:

```
tensor([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,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Feed-forward architectures replicating a logistic regression or with one hidden layer are similar to those in Sect. 8.5.3. We have an output of three classes in the last layer and no activation. The reason is that PyTorch's cross entropy loss computes it automatically from logits (the last matrix output).

The logistic regression model consists of one linear layer:

```
input_dim = X_scaled.size(dim=1)
model = nn.Sequential(nn.Linear(input_dim, 3))
```

We create a model with one hidden layer of five nodes by adding a linear module followed by a *ReLU* activation:

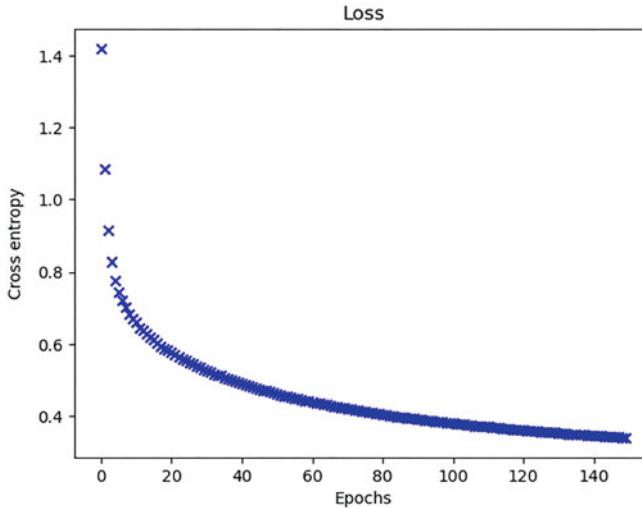
```
from collections import OrderedDict

model = nn.Sequential(OrderedDict([
    ('W1', nn.Linear(input_dim, 5)),
    ('ReLU', nn.ReLU()),
    ('W2', nn.Linear(5, 3))
]))
```

As in Sect. 8.5.3, we define a loss, this time `nn.CrossEntropyLoss()`, and an optimizer:

```
loss_fn = nn.CrossEntropyLoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

By default, PyTorch cross entropy reduces the loss of each mini-batch with its mean. In the next steps, we will compute the loss mean for the whole dataset. It is then preferable to sum the loss of each mini-batch, sum all the mini-batches, and then compute the mean.



**Fig. 8.7** The training loss over the epochs for the logistic regression model

The dataloader is the same as in Sect. 8.5.3:

```
BATCH_SIZE = 4
dataset = TensorDataset(X_scaled, y)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

The fitting loop is also similar and we record the loss. See Fig. 8.7.

```
model.train()
ce_loss = []
for epoch in range(150):
    loss_train = 0
    for X_scaled_batch, y_batch in dataloader:
        y_batch_pred = model(X_scaled_batch)
        loss = loss_fn(y_batch_pred, y_batch)
        loss_train += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    ce_loss += [loss_train/len(y)]
```

We apply the model to the dataset to see the predicted values:

```
model.eval()

with torch.no_grad():
    Y_pred_logits = model(X_scaled)
```

We obtain the result below for the logistic regression model. It is not normalized as the last layer has no activation:

```
tensor([[ 1.2929, -1.4799,  0.4301],
       [ 0.9736,  0.2044, -0.9725],
       [ 0.9331,  0.4184, -1.1506],
       ...]])
```

Logits are difficult to interpret and we apply a softmax function to the rows:

```
with torch.no_grad():
    Y_pred_proba = torch.softmax(model(X_scaled), dim=-1)
```

so that we have probabilities:

```
tensor([[0.6736, 0.0421, 0.2843],
       [0.6226, 0.2885, 0.0889],
       [0.5807, 0.3471, 0.0723],
       ...]])
```

We finally extract the predicted classes with argmax:

```
y_pred = torch.argmax(Y_pred_proba, dim=-1)
```

The result is the same as with Keras:

```
tensor([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,
       2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2])
```

Note again that a serious evaluation would use a separate dataset.

## 8.9 Backpropagation in PyTorch

The training loop of PyTorch that we first introduced in Sect. 8.5.3 is more complex and more difficult to understand than the fitting functions of both Keras and scikit-learn. In addition, the connection with gradient descent (Sect. 7.5) and the mathematical equations of backpropagation (Sect. 8.3) is not obvious at first sight. We will walkthrough this loop with the 5-hidden node network from the previous section, where we used a learning rate of 0.01 and we will see how PyTorch implements all this.

Let us examine the model after we have run training epochs and suppose that we are at step  $k$ . We will focus on the first layer  $W^{(1)}$  consisting of five nodes. All the other parameters would show the same behavior. We print its weight matrix with:

```
model.W1.weight
```

yielding:

```
tensor([[-0.8995,  0.3332],
       [ 0.9948, -1.4310],
       [-0.2389,  0.5856],
       [-1.9805,  2.0227],
       [-0.6628,  0.2297]], requires_grad=True)
```

This represents the state of the model at a certain iteration, here step  $k$ . Using this model, we can predict the targets and measure the difference with the true

values with the cross entropy loss, here reduced as a sum. This corresponds to the statement:

```
loss = loss_fn(model(X_scaled), y)
# tensor(10.3754, grad_fn=<NllLossBackward0>)
```

The tensors store gradients when `requires_grad` is true. We print the gradient of  $W^1$  with:

```
model.W1.weight.grad
```

yielding:

```
tensor([[-0.0000,  0.0000],
       [-0.0000,  0.0000],
       [ 0.0028, -0.0114],
       [-0.0000,  0.0000],
       [ 0.0039, -0.0158]])
```

This gradient comes from an earlier operation. We saw that PyTorch accumulates them. Before we backpropagate the gradients of the current loss, we need to clear the old ones with:

```
optimizer.zero_grad()
```

and calling again

```
model.W1.weight.grad
```

prints nothing. We can now backpropagate the gradients safely to all the layers starting from the last one with:

```
loss.backward()
```

$W^{(1)}$  has a new gradient corresponding to the current loss

```
>>> model.W1.weight.grad
tensor([[ 0.1993, -0.1766],
       [-0.0467,  0.0465],
       [ 0.0617, -0.0510],
       [ 0.5730, -0.4875],
       [ 0.0860, -0.0710]])
```

We can now update the weight parameters with the relation:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla f(\mathbf{w}_k)$$

to have their values at step  $k + 1$ . This corresponds to the statement:

```
>>> model.W1.weight = 0.01 * model.W1.weight.grad
tensor([[-0.9015,  0.3350],
       [ 0.9953, -1.4314],
       [-0.2395,  0.5861],
       [-1.9862,  2.0276],
       [-0.6636,  0.2304]], grad_fn=<SubBackward0>)
```

In the training loop, we applied the updates with the statement:

```
>>> optimizer.step()
```

that yields exactly the same weight values as our hand-calculations

```
>>> model.W1.weight
tensor([[-0.9015,  0.3350],
       [ 0.9953, -1.4314],
       [-0.2395,  0.5861],
       [-1.9862,  2.0276],
       [-0.6636,  0.2304]], requires_grad=True)
```

## 8.10 Further Reading

Neural networks and gradient descent are described in a countless number of books. As in the previous chapter, Murphy (2022), James et al. (2021), and Goodfellow et al. (2016) are good starting points. In many applications, the parameter values of gradient descent are very important to the success of the training procedures. Among these parameters, the optimizer has a considerable influence on the result. Ruder (2017) is a good survey of available gradient descent algorithms.

A number of machine-learning toolkits are available from the Internet. Scikit-learn<sup>2</sup> (Pedregosa et al. 2011) is a comprehensive collection of machine-learning and data analysis algorithms with an API in Python. Keras<sup>3</sup> (Chollet 2021) is a neural network library also in Python. PyTorch<sup>4</sup> is another one, which is a conversion in Python of Torch, an older library written in Lua. R<sup>5</sup> is another set of statistical and machine-learning functions with a script language.

We used scikit-learn, Keras and PyTorch in this chapter. Chollet (2021) is an excellent and pedagogical tutorial on Keras and machine learning in general. Stevens et al. (2020) is a book by programmers who participated in the development of PyTorch. Raschka et al. (2022) is a third one that covers both PyTorch and scikit-learn.

---

<sup>2</sup> <https://scikit-learn.org/>.

<sup>3</sup> <https://keras.io/>.

<sup>4</sup> <https://pytorch.org/>.

<sup>5</sup> <https://www.r-project.org/>.

# Chapter 9

## Counting and Indexing Words



### 9.1 Text Segmentation

Many language processing techniques rely on words and sentences. When this is the case and 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 may also be preceded or include a cleaning step to remove formatting instructions, such as XML tags, if any, and a normalization with NFC or NFKC, see Sect. 4.1.3.

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. 9.1).

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.

#### 9.1.1 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.

**Fig. 9.1** Latin inscriptions on the *lapis niger*. *Corpus inscriptionum latinarum*, CIL I. Picture from Wikipedia



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 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 eighteenth–nineteenth century), 9 812,345 (current French and German) 9.812,345 (French from the nineteenth–early twentieth 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.

### 9.1.2 *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', ':']]
```

A 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
:
```

## 9.2 Tokenizing Words

We now introduce two complementary tokenization techniques:

1. The first technique matches the token content between the delimiters;
2. The second one identifies the boundaries, where it splits the input string.

Both use the white spaces to delimit the words. We will use the first lines of the *Odyssey* as example:

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

and we assign them 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."""
```

As our patterns will include Unicode classes, we import the `regex` module:

```
import regex as re
```

We will use the letters, `\p{L}`, numbers, `\p{N}`, punctuation, `\p{P}`, and symbols, `\p{S}`; see Table 4.6 for a complete list of the classes.

### 9.2.1 Defining Content

We can first define the words as sequences of contiguous letters with the `\p{L}+` pattern. The tokenization is straightforward in Python with the `re.findall()` function that returns a list of words:

```
>>> re.findall(r'\p{L}+', text)
['Tell', 'me', 'O', 'muse', 'of', 'that', 'ingenious', 'hero',
'who', 'travelled', 'far', 'and', 'wide', ..., 'of', 'Troy']
```

This regex is very simple, but it ignored the punctuation as well as the other character sequences. If we want to extract the words and the rest, excluding the spaces, we need to add another pattern to specify the nonword tokens:

```
r'\p{L}+|[^s\p{L}]+'
```

This disjunction creates a token from a letter sequence (`\p{L}+`) as well as from a sequence of other characters (`[^s\p{L}]+`):

```
>>> re.findall(r'\p{L}+|[^s\p{L}]+', text)
['Tell', 'me', ',', 'O', 'muse', ',', 'of', 'that',
'ingenious', 'hero', 'who', 'travelled', ..., 'Troy', '.']
```

If we want to create separate tokens for the numbers, we add `\p{N}+`:

```
r'\p{L}+|\p{N}+|[^s\p{L}\p{N}]+'
```

Finally, this regex:

```
r'\p{L}+|\p{N}+|\p{P}|[^s\p{L}\p{N}\p{P}]+'
```

tokenizes the punctuation separately and creates a single token for each punctuation sign.

### 9.2.2 Using Boundaries

The second technique matches the word delimiters to tokenize a text. In its simplest form, we just split the text when we encounter a sequence of white spaces and we return the tokens between the delimiters in a list.

Python has a `split()` function that can just do that. It 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 will use `re.split()` as it is more flexible and the pattern

```
pattern1 = r'\s+'
```

to represent a sequence of white spaces. Running:

```
re.split(pattern1, text)
```

we obtain:

```
[‘Tell’, ‘me’, ‘O’, ‘muse’, ‘of’, ‘that’, ‘ingenious’,  
‘hero’, ‘who’, ‘travelled’, ‘far’, ..., ‘of’, ‘Troy.’]
```

where the commas are not segmented from the words.

To tokenize them, we need to process separately the punctuation and symbols. We identify these characters with the class `[\p{S}\p{P}]`. We then use a substitution to insert one space before and after them:

```
pattern2 = r'([\p{S}\p{P}]+)'  
re.sub(pattern2, r' \1 ', text)
```

As a result, we have separated the punctuation from the words. We then tokenize the text according to white spaces as in the previous segmentation. Altogether, we have this function:

```
re.split(  
    pattern1,  
    re.sub(pattern2, r' \1 ', text))
```

Running this code on our small text results in:

```
[‘Tell’, ‘me’, ‘’, ‘o’, ‘muse’, ‘’, ‘of’, ‘that’,
‘ingenious’, ‘hero’, ‘who’, …, ‘Troy’, ‘.’, ‘’]
```

This `re.split()` produces empty strings. We filter them out from our token list with the statement:

```
filter(None, token_list)
```

where `None` acts as the identity function. As we have seen, this second technique is a bit more convoluted than the direct identification of words. Most of the time, the first technique is to be preferred.

### 9.2.3 Improving Tokenization

The tokenizing programs we have created so far are not yet perfect. Decimal numbers, for example, would not be properly processed. They would match the point of decimal numbers such as 3.14 and create the tokens 3 and 14. The apostrophe inside words is another ambiguous sign. The tokenization of auxiliary and negation contractions in English would then need a morphological analysis.

Improving the tokenizers requires more complex regular expressions that would take into account word forms such as those in Table 9.1.

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. This would also require a more elaborate regular expression.

**Table 9.1** Apostrophe tokenization in English

Contracted form	Example	Tokenization	Expanded form
’m	<i>I’m</i>	I ’m	<i>I am</i>
’d	<i>we’d</i>	we ’d	<i>we had or we would</i>
’ll	<i>we’ll</i>	we ’ll	<i>we will</i>
’re	<i>you’re</i>	you ’re	<i>you are</i>
’ve	<i>I’ve</i>	I ’ve	<i>I have</i>
n’t	<i>can’t</i>	can n’t	<i>cannot</i>
’s	<i>she’s</i>	she ’s	<i>she has or she is</i>
’s	<i>Pierre’s book</i>	Pierre ’s book	Possessive marking

### 9.2.4 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. 7.9) 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.

Once we have an annotation, we need to define the features we will use for the classifier. We already used features in Sect. 7.1 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 9.2 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.*

**Table 9.2** 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**

Context	Current char.	Previous pair	Next char.	Next pair	Class	Action
<i>Vinken</i> ,	<i>n</i>	<i>en</i>	,	,	Token end	New line
<i>old</i> ,	<i>d</i>	<i>ld</i>	,	,	Token end	New line
<i>Nov.</i>	<i>v</i>	<i>ov</i>	.	.	Inside token	Nothing

From these features, the classifier will create a 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 or Universal Dependencies (Nivre et al. 2017) for more than 150 languages, including English.

We extract a training dataset 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.

## 9.3 Sentence Segmentation

### 9.3.1 *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.

### 9.3.2 *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

**Table 9.3** Recognizing numbers. After Grefenstette and Tapanainen (1994)

Fractions, dates	$[0-9]+(\backslash/[0-9]+)+$
Percent	$([+\backslash-])?[0-9]+(\backslash.)?[0-9]*$
Decimal numbers	$([0-9]+,?) + (\backslash. [0-9]+   [0-9]+)*$

different processing stages. However, there will remain cases where the program fails, notably with abbreviations.

### 9.3.3 Using Regular Expressions

Starting from the most simple rule to identify 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 9.3 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 9.3 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 9.4 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%.

### 9.3.4 Improving the Segmenter Using Lexicons

Grefenstette and Tapanainen (1994) further improved their segmenter 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

**Table 9.4** 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]\.	1327	52	14
[A-Za-z]\.([A-Za-z0-9]\.)+	570	0	66
[A-Z][bcdfghj-np-tvxz]+\.	1938	44	26
Totals	3835	96	106

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.

### 9.3.5 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. 9.2.4 and we can use the same corpus: we just ignore the <SPLIT> tags.

Practically, we need to collect a dataset 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 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 9.5** The features extracted from *Nov.* and 29. in the example sentences in Sect. 9.2.4. The two classes to learn are **inside sentence** and **end of sentence**

Context	Prefix	Suffix	Previous word	Next word	Prefix abbrev.	Class
<i>Nov.</i>	<i>Nov</i>	nil	<i>director</i>	29.	Yes	Inside sentence
29.	29	nil	<i>Nov.</i>	<i>Mr.</i>	No	End of sentence

Table 9.5 shows the features for the periods in *Nov.* and 29. in the example sentences in Sect. 9.2.4. 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. 9.3.4.

Reynar and Ratnaparkhi (1997) used logistic regression to train their classification models and discriminate between the two classes: **inside sentence** and **end of sentence**.

## 9.4 Word Counting

### 9.4.1 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.

### 9.4.2 Counting Words with Python

Extracting and counting words is straightforward and very fast with Python. We can obtain them with the following algorithm:

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. 2.6.9. Let us use functions to implement the different steps. For the first step, we apply a tokenizer

to the text (Sect. 9.2) 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(r'\p{L}+', text)
    return words

def count_words(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)
    word_freqs = count_words(words)
    for word in sorted(word_freqs()):
        print(word, '\t', word_freqs[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. 2.6.9 that we have to assign the `key` argument the value `word_freqs.get` in `sorted()`. We also set the `reverse` argument to `True`:

```
sorted(word_freqs.keys(), key=word_freqs.get, reverse=True)
```

Our program is now ready and we can apply it to a corpus. Let us try it with Homer's *Iliad*, where an English version translated by Samuel Butler is available at the URL <https://classics.mit.edu/Homer/iliad.mb.txt>. We download the file with the statements:

```
import requests
text_copyright = requests.get(
    'http://classics.mit.edu/Homer/iliad.mb.txt').text
```

The full text contains a copyright information that we want to exclude from the counts. It consists of a header and a footer separated from the main text by a dashed line. We extract the narrative with the regex:

```
text = re.search(r'^-+$(.+)^-+$',
                text_copyright, re.M | re.S).group(1).strip()
```

Running the program on this text and limiting the output to 10 most frequent words, we obtain:

```
the    9948
and   6624
of    5606
to    3329
he    2905
his   2537
in    2242
him   1868
you   1810
a     1807
```

### 9.4.3 The Counter Class

Counting words is a very frequent operation in text processing. In the previous section, we wrote a function to do it. We already saw in Sect. 2.16.3 that Python has also a dedicated `Counter` class for this that will save us time. We create a `Counter` object with the list of words as argument:

```
from collections import Counter

words = tokenize(text)
word_freqs = Counter(words)
```

As in our initial counting function, a `Counter` object is a dictionary, where the keys are the words and the values, the frequencies.

```
>>> word_freqs['hector']
480
```

In Sect. 2.16.3, we also saw that, differently to plain dictionaries, accessing a missing key does not throw an exception:

```
>>> word_freqs['computer']
0
```

but it does not create the key in the counter:

```
>>> 'computer' in word_freqs
False
```

The  $n$  most common words are the same as with our own dictionary:

```
>>> word_freqs.most_common(5)
[('the', 9948),
 ('and', 6624),
 ('of', 5606),
 ('to', 3329),
 ('he', 2905)]
```

#### 9.4.4 A Crash Program To Count Words with Unix

Finally, let us use Unix tools to count words. 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.

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.<sup>1</sup> We reproduce it here (slightly modified):

1. `tr -cs 'A-Za-z' '\n' <input_file |`

Tokenize the text in `input_file` with one word per line. This transliteration command is the most complex of the six.

(a) `tr` has two arguments `search_list` and `replacement_list`. It replaces all the occurrences of the characters in `search_list` by the corresponding character in `replacement_list`: The first character in `search_list` by the first one in `replacement_list` and so on. For instance, the instruction `tr 'ABC' 'abc'` replaces the occurrences of  $A$ ,  $B$ , and  $C$  by  $a$ ,  $b$ , and  $c$ , respectively.

(b) `tr` has a few options:

- `d` deletes any characters of the search list that have no corresponding character in the replacement list;
- `c` translates characters that belong to the complement of the search list. After complementing the search list, if the replacement list has not the same length, the last character of this replacement list is repeated so that we have equal lengths;
- `s` reduces—squeezes, squashes—sequences of characters translated to an identical character to a single instance.

The command

`tr -d 'AEIOUaeiou'`

deletes all the vowels and

`tr -cs 'A-Za-z' '\n'`

replaces all nonletters with a new line. The contiguous sequences of translated new lines are reduced to a single one.

The `tr` output is passed to the next command.

---

<sup>1</sup> Stephen Bourne, the author of the Unix Bourne shell, proposed a similar script; see Bourne (1982, pp. 196–197).

**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
```

## 9.5 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 as of today.

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.

### 9.5.1 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

**Table 9.6** 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
<i>America</i>	(D1, 7)
<i>Chrysler</i>	(D1, 1) → (D2, 1)
<i>In</i>	(D1, 5) → (D2, 5)
<i>Investments</i>	(D1, 4) → (D2, 4)
<i>Latin</i>	(D1, 6)
<i>Major</i>	(D2, 3)
<i>Mexico</i>	(D2, 6)
<i>New</i>	(D1, 3)
<i>Plans</i>	(D1, 2) → (D2, 2)

the corresponding document. The resulting data structure is called an *inverted index* and Table 9.6 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.

### 9.5.2 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:

```
{
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. 9.2.1, 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.findall(r'\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.<sup>2</sup>

```
>>> 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]
```

Finally, we collect of collection of books by Dickens and 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]

```

---

<sup>2</sup> Retrieved from the Gutenberg Project, [www.gutenberg.org](http://www.gutenberg.org), November 3, 2016.

Applying this program 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]}
```

### 9.5.3 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  $D$  would be represented as:

$$\mathbf{d} = (C(w_1), C(w_2), C(w_3), \dots, C(w_n)).$$

Table 9.7 shows the document vectors representing the examples in Sect. 9.5.1, and Table 9.8 shows a general matrix representing a collection of documents, where each cell  $(w_i, D_j)$  contains the frequency of  $w_i$  in document  $D_j$ .

**Table 9.7** The vectors representing the two documents in Sect. 9.5.1. The words have been normalized in lowercase letters

D#\backslash 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 9.8** The word by document matrix. Each cell  $(w_i, D_j)$  contains the frequency of  $w_i$  in document  $D_j$

D#\backslash Words	$w_1$	$w_2$	$w_3$	...	$w_m$
$D_1$	$C(w_1, D_1)$	$C(w_2, D_1)$	$C(w_3, D_1)$	...	$C(w_m, D_1)$
$D_2$	$C(w_1, D_2)$	$C(w_2, D_2)$	$C(w_3, D_2)$	...	$C(w_m, D_2)$
...					
$D_n$	$C(w_1, D_n)$	$C(w_2, D_n)$	$C(w_3, D_n)$	...	$C(w_m, D_n)$

Using the vector space model, we can measure the similarity between two documents,  $D$  and  $Q$ , by the angle they form in the vector space. In practice, it is easier to compute the cosine of the angle between  $\mathbf{q}$ , representing  $Q$ , and  $\mathbf{d}$ , defined as:

$$\cos(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{||\mathbf{q}|| \cdot ||\mathbf{d}||} = \frac{\sum_{i=1}^n q_i d_i}{\sqrt{\sum_{i=1}^n q_i^2} \sqrt{\sum_{i=1}^n d_i^2}}.$$

The cosine values will range from 0, meaning very different documents, to 1, very similar or identical documents.

#### 9.5.4 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-idf* or *tf × 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-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-idf* that take into account the document length.

### 9.5.5 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.

## 9.6 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.

### 9.6.1 Corpora

The corpora used in text categorization vary depending of the applications. In sentiment analysis (or opinion mining), the goal is to classify judgments or emotions expressed, for instance, in product reviews collected from consumer forums. The annotations use three base categories: positive, negative, or neutral. The IMDB corpus consisting of movie reviews is a frequently cited example for such a task (Maas et al. 2011); in spam detection, the categorizer classifies electronic messages into two classes: *spam* or *no spam*. The SpamAssassin public mail corpus is an early collection of messages annotated with such labels.

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.

### 9.6.2 Building a Categorizer with Scikit-Learn

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. 7.1). The training procedure uses a bag-of-word representation of the documents, either with Boolean features, term frequencies, or  $tf \times idf$  as input, and their classes as output.

Logistic regression again is a simple, yet efficient technique to carry out text classification. LibShortText (Yu et al. 2013), for example, is an open source library that includes logistic regression and different types of preprocessing and feature representations.

In this section, we will build a simple text categorizer using scikit-learn and its built-in modules. As corpus, we will use Homer's *Iliad* and *Odyssey*, where each work consists of 24 books. We will consider that each book forms a document.

The first step is to load each book as a string and store it in a list that we call `homer_corpus`. We also store the work names, either `iliad` or `odyssey`, as strings in a list called `homer_titles`.

The statement `homer_corpus[0] [:60]` returns 60 characters from the first book in the list:

```
Book I\n\nTHE GODS IN COUNCIL--MINERVA'S VISIT TO ITHACA--THE
```

and `homer_titles` contains the works the books come from:

```
['odyssey', 'odyssey', 'odyssey', 'odyssey', 'odyssey', ...
 ..., 'iliad', 'iliad', 'iliad', 'iliad', 'iliad']
```

Before we train a model, we split the corpus into training and test sets using the `train_test_split()` function from scikit-learn. As the corpus is small, we set the size of the test set to be 20%:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    homer_corpus,
    homer_titles,
    test_size=0.20)
```

The `homer_corpus` list contains strings. We need to convert them to numerical representations. To do this, we apply two transformations that produce bag-of-word representations of the documents, both using scikit-learn functions. The first one `CountVectorizer()` results in vectors of word counts and the second one, `TfidfTransformer()`, in tf-idf parameters:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer

count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(X_train)

tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
```

The resulting matrix, `X_train_tfidf`, has a shape of (38, 8833), meaning that we have 38 books, the documents, about 80% of the corpus, and rows of 8833 words representing each book.

We train a model as in Sect. 7.11.1:

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression().fit(X_train_tfidf, y_train)
```

and we evaluate it on the test set.

Before applying a prediction, we must convert the test documents into numerical vectors:

```
X_test_counts = count_vect.transform(X_test)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)
```

Then we predict the test set and measure the accuracy score:

```
from sklearn.metrics import accuracy_score

y_test_hat = clf.predict(X_test_tfidf)
accuracy_score(y_test, y_test_hat)
```

In this small experiment, we obtain a 100% accuracy, probably because the names are different in the *Iliad* and the *Odyssey*.

## 9.7 Further Reading

Tokenization and sentence segmentation are very frequent operations in NLP. Although essential, they are merely considered technical details and they have not gathered much scientific interest. Nonetheless, there are several toolkits that provide functions to carry out tokenization and sentence detection:

1. The Natural Language Toolkit (NLTK)<sup>3</sup> and spaCy<sup>4</sup> are popular open-source libraries in Python. They provides a set of tokenizers, segmenters, as well as document categorizers;
2. Apache OpenNLP<sup>5</sup> is a complete suite of NLP modules in Java that also includes, *inter alia*, a sentence detector, a tokenizer, and a document categorizer.

It is difficult to develop a multilingual tokenizer based on rules because of the specificities of each language. Statistical or machine-learning methods do not require expert knowledge but only segmented corpora. Shao et al. (2018) created such a machine-learning based tokenizer that the authors trained on the Universal Dependencies corpora.

The indexing, 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 more complete reviews of techniques on indexing and information retrieval, see Manning et al. (2008) or Baeza-Yates and Ribeiro-Neto (2011).

The comparison of a query with documents vectorized with tf-idf involves the computation of as many cosines as there are documents. This is prohibitive for large corpora. A solution to speed up this process is to store the document vectors in a

---

<sup>3</sup> <https://www.nltk.org/>.

<sup>4</sup> <https://spacy.io/>.

<sup>5</sup> <https://opennlp.apache.org/>.

vector database and use approximate comparison algorithms such as Faiss (Johnson et al. 2019).

Lucene<sup>6</sup> is a popular open-source indexer written in Java with a Python API. It is used in scores of web sites such as Twitter and Wikipedia to carry out search and information retrieval.

---

<sup>6</sup> <https://lucene.apache.org/>.

# Chapter 10

## 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. 10.4.2.

### 10.1 Modeling Word Sequences

We saw in Chap. 3, *Corpus Processing Tools* 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 beyond speech recognition and lexicography. 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.

## 10.2 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 10.1 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*.

**Table 10.1** Ranking and generating words using trigrams. After Jelinek (1990)

Word	Rank	More likely alternatives
<i>We</i>	9	<i>The This One Two A Three Please In</i>
<i>need</i>	7	<i>are will the would also do</i>
<i>to</i>	1	
<i>resolve</i>	85	<i>have know do...</i>
<i>all</i>	9	<i>the this these problems...</i>
<i>of</i>	2	<i>the</i>
<i>the</i>	1	
<i>important</i>	657	<i>document question first...</i>
<i>issues</i>	14	<i>thing point to...</i>
<i>within</i>	74	<i>to of and in that...</i>
<i>the</i>	1	
<i>next</i>	2	<i>company</i>
<i>two</i>	5	<i>page exhibit meeting day</i>
<i>days</i>	5	<i>weeks years pages months</i>

### 10.2.1 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. 2.6.7) 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 program is nearly identical to that of Sect. 9.4.2 for unigrams. As input, we uses the same list of words and `Counter` to count the items in the list:

```
words = tokenize(text.lower())
bigrams = [tuple(words[idx:idx + 2])
           for idx in range(len(words) - 1)]
bigram_freqs = Counter(bigrams)
```

The five most frequent bigrams of the *the Iliad* are:

```
>>> bigram_freqs.most_common(5)
```

```
[(('of', 'the'), 1239),
 ('son', 'of'), 825),
 ('to', 'the'), 582),
 ('and', 'the'), 518),
 ('in', 'the'), 500)]
```

The bigram count can easily be generalized to  $n$ -grams with the statement:

```
ngrams = [tuple(words[idx:idx + n])
           for idx in range(len(words) - n + 1)]
```

Setting  $n = 3$ , we obtain the five most frequent trigrams from the *the Iliad* with:

```
>>> Counter(ngrams).most_common(5)
```

```
[(('the', 'son', 'of'), 354),
 ('of', 'the', 'achaeans'), 203),
 ('son', 'of', 'atreus'), 118),
 ('son', 'of', 'peleus'), 98),
 ('the', 'trojans', 'and'), 94)]
```

### 10.2.2 Counting Bigrams with Unix

With the Unix tools from Sect. 9.4.4, it is also easy to extend the counts from unigrams 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.

## 10.3 Probabilistic Models of a Word Sequence

### 10.3.1 The Maximum Likelihood Estimation

We observed in Table 10.1 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  $S = w_1, w_2, \dots, w_i, \dots, w_n$  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  $w_1, w_2, \dots, w_i, \dots, w_n$  in the corpus. This estimate is called the *maximum likelihood estimate* (MLE):

$$P_{\text{MLE}}(S) = \frac{C(w_1, \dots, w_n)}{N},$$

where  $C(w_1, \dots, w_n)$  is the frequency or count of the string  $w_1, w_2, \dots, w_i, \dots, w_n$  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:

$$\begin{aligned} P(S) &= P(w_1, \dots, w_n), \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\dots P(w_n|w_1, \dots, w_{n-1}), \\ &= \prod_{i=1}^n P(w_i|w_1, \dots, w_{i-1}). \end{aligned}$$

The probability  $P(\text{It, was, a, bright, cold, day, in, April})$  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:

$$\begin{aligned} P(S) = & P(\text{It}) \times P(\text{was}|\text{It}) \times P(a|\text{It}, \text{was}) \times P(\text{bright}|\text{It}, \text{was}, a) \times \dots \\ & \times P(\text{April}|\text{It}, \text{was}, a, \text{bright}, \dots, \text{in}). \end{aligned}$$

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:

$$P(w_i|w_1, w_2, \dots, w_{i-1}) \approx P(w_i|w_{i-1}),$$

or trigrams:

$$P(w_i|w_1, w_2, \dots, w_{i-1}) \approx P(w_i|w_{i-2}, w_{i-1}).$$

Using a trigram language model,  $P(S)$  is approximated as:

$$\begin{aligned} P(S) \approx & P(\text{It}) \times P(\text{was}|\text{It}) \times P(a|\text{It}, \text{was}) \times P(\text{bright}|\text{was}, a) \times \dots \\ & \times P(\text{April}|\text{day}, \text{in}). \end{aligned}$$

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:

$$\begin{aligned} P_{\text{MLE}}(w_{i+n}|w_{i+1}, \dots, w_{i+n-1}) &= \frac{C(w_{i+1}, \dots, w_{i+n})}{\sum_w C(w_{i+1}, \dots, w_{i+n-1}, w)}, \\ &= \frac{C(w_{i+1}, \dots, w_{i+n})}{C(w_{i+1}, \dots, w_{i+n-1})}. \end{aligned}$$

As the probabilities we obtain are usually very low, it is safer to represent them as a sum of logarithms in practical applications. For the bigrams, we will then use:

$$\log P(S) \approx \log P(w_1) + \sum_{i=2}^n \log P(w_i|w_{i-1}),$$

or the negative log-likelihood (NLL):

$$NLL(P(S)) \approx -\log P(w_1) - \sum_{i=2}^n \log P(w_i|w_{i-1}),$$

instead of  $P(S)$ . Nonetheless, in the following sections, as our example corpus is very small, we will compute the probabilities using products.

### 10.3.2 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. Like in our machine-learning experiments, 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 **validation set**, also called **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 tenfold cross-validation if we repeat it ten times. Cross-validation smoothes the impact of a specific partition of the corpus; see Sect. 6.4.3.

## 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:

*<ss> A good deal of the literature of the past was, indeed, already being transformed in this way </ss>*

We first normalized the text: we created a file with one sentence per line. We inserted automatically the delimiters `<ss>` and `</ss>`. 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 10.2 shows the unigram and bigram frequencies for the words of the test sentence.

**Table 10.2** Frequencies of unigrams and bigrams. We excluded the `<ss>` symbols from the word counts

$w_i$	$C(w_i)$	#words	$P_{\text{MLE}}(w_i)$	$w_{i-1}, w_i$	$C(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{MLE}}(w_i   w_{i-1})$
<code>&lt;ss&gt;</code>	7072	—	—	—	—		
<i>a</i>	2482	108,140	0.023	<code>&lt;ss&gt; a</code>	133	7072	0.019
<i>good</i>	53	108,140	0.00049	<i>a good</i>	14	2482	0.006
<i>deal</i>	5	108,140	$4.62 \times 10^{-5}$	<i>good deal</i>	0	53	0.0
<i>of</i>	3310	108,140	0.031	<i>deal of</i>	1	5	0.2
<i>the</i>	6248	108,140	0.058	<i>of the</i>	742	3310	0.224
<i>literature</i>	7	108,140	$6.47 \times 10^{-5}$	<i>the literature</i>	1	6248	0.00016
<i>of</i>	3310	108,140	0.031	<i>literature of</i>	3	7	0.429
<i>the</i>	6248	108,140	0.058	<i>of the</i>	742	3310	0.224
<i>past</i>	99	108,140	0.00092	<i>the past</i>	70	6248	0.011
<i>was</i>	2211	108,140	0.020	<i>past was</i>	4	99	0.040
<i>indeed</i>	17	108,140	0.00016	<i>was indeed</i>	0	2211	0.0
<i>already</i>	64	108,140	0.00059	<i>indeed already</i>	0	17	0.0
<i>being</i>	80	108,140	0.00074	<i>already being</i>	0	64	0.0
<i>transformed</i>	1	108,140	$9.25 \times 10^{-6}$	<i>being transformed</i>	0	80	0.0
<i>in</i>	1759	108,140	0.016	<i>transformed in</i>	0	1	0.0
<i>this</i>	264	108,140	0.0024	<i>in this</i>	14	1759	0.008
<i>way</i>	122	108,140	0.0011	<i>this way</i>	3	264	0.011
<code>&lt;/ss&gt;</code>	7072	108,140	0.065	<i>way &lt;/ss&gt;</i>	18	122	0.148

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:

$$\begin{aligned} P(S) &\approx P(a) \times P(good) \times \dots \times P(way) \times P(</s>), \\ &\approx 3.67 \times 10^{-48}. \end{aligned}$$

As  $P(</s>)$  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:

$$P(S) \approx P(a|</s>) \times P(good|a) \times \dots \times P(way|this) \times P(</s>|way).$$

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.

## 10.4 Smoothing $N$ -Gram Probabilities

### 10.4.1 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.

### 10.4.2 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.

**Table 10.3** Frequencies of bigrams using Laplace's rule

$w_{i-1}, w_i$	$C(w_{i-1}, w_i) + 1$	$C(w_{i-1}) +  V $	$P_{\text{Lap}}(w_i   w_{i-1})$
$\langle s \rangle a$	$133 + 1$	$7072 + 8635$	0.0085
<i>a good</i>	$14 + 1$	$2482 + 8635$	0.0013
<i>good deal</i>	$0 + 1$	$53 + 8635$	0.00012
<i>deal of</i>	$1 + 1$	$5 + 8635$	0.00023
<i>of the</i>	$742 + 1$	$3310 + 8635$	0.062
<i>the literature</i>	$1 + 1$	$6248 + 8635$	0.00013
<i>literature of</i>	$3 + 1$	$7 + 8635$	0.00046
<i>of the</i>	$742 + 1$	$3310 + 8635$	0.062
<i>the past</i>	$70 + 1$	$6248 + 8635$	0.0048
<i>past was</i>	$4 + 1$	$99 + 8635$	0.00057
<i>was indeed</i>	$0 + 1$	$2211 + 8635$	0.000092
<i>indeed already</i>	$0 + 1$	$17 + 8635$	0.00012
<i>already being</i>	$0 + 1$	$64 + 8635$	0.00011
<i>being transformed</i>	$0 + 1$	$80 + 8635$	0.00011
<i>transformed in</i>	$0 + 1$	$1 + 8635$	0.00012
<i>in this</i>	$14 + 1$	$1759 + 8635$	0.0014
<i>this way</i>	$3 + 1$	$264 + 8635$	0.00045
<i>way </i> $\langle /s \rangle$	$18 + 1$	$122 + 8635$	0.0022

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: 5000 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:

$$P_{\text{Laplace}}(w_i | w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{\sum_w (C(w_{i-1}, w) + 1)} = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + |V|},$$

where  $|V|$  is the cardinality of the vocabulary, i.e. 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 10.3):

$$\begin{aligned} P_{\text{Laplace}}(S) &\approx P(a | \langle s \rangle) \times P(\text{good} | a) \times \dots \times P(\langle /s \rangle | \text{way}), \\ &\approx 4.62 \times 10^{-57}. \end{aligned}$$

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  $\lambda$ . The new formula is then:

$$P_{\text{Lidstone}}(w_i | w_{i-1}) = \frac{C(w_{i-1}, w_i) + \lambda}{C(w_{i-1}) + \lambda|V|},$$

which, however, is not a big improvement.

### 10.4.3 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  $|V|^2$  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  $2 \times (5820/37,365) = 0.31$ . Table 10.4 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:

$$P_{\text{GT}}(w_1, \dots, w_n) = \frac{c^*(w_1, \dots, w_n)}{N},$$

where  $c^*$  is the reestimated count of  $w_1 \dots w_n$ , and  $N$  the original count of  $n$ -grams in the corpus. The conditional frequency is

$$P_{\text{GT}}(w_n | w_1, \dots, w_{n-1}) = \frac{c^*(w_1, \dots, w_n)}{C(w_1, \dots, w_{n-1})}.$$

Table 10.5 shows the conditional probabilities, where only frequencies less than 10 have been reestimated. The sentence probability using bigrams is  $2.56 \times 10^{-50}$ . This is better than with Laplace’s rule, but as the corpus is very small, still greater than the unigram probability.

**Table 10.4** The reestimated frequencies of the bigrams

	Frequency of occurrence	$N_c$	$c^*$
0		74,513,701	0.0005
1		37,365	0.31
2		5820	1.09
3		2111	2.02
4		1067	3.37
5		719	3.91
6		468	4.94
7		330	6.06
8		250	6.44
9		179	8.94

**Table 10.5** The conditional frequencies using the Good–Turing method. We have not reestimated the frequencies when they are greater than 9

$w_{i-1}, w_i$	$C(w_{i-1}, w_i)$	$c^*(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{GT}}(w_i w_{i-1})$
$\langle s \rangle a$	133	133	7072	0.019
$a good$	14	14	2482	0.006
$good deal$	0	0.0005	53	$9.46 \times 10^{-6}$
$deal of$	1	0.31	5	0.062
$of the$	742	742	3310	0.224
$the literature$	1	0.31	6248	$4.99 \times 10^{-5}$
$literature of$	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	$2.27 \times 10^{-7}$
$indeed already$	0	0.0005	17	$2.95 \times 10^{-5}$
$already being$	0	0.0005	64	$7.84 \times 10^{-6}$
$being transformed$	0	0.0005	80	$6.27 \times 10^{-6}$
$transformed in$	0	0.0005	1	0.00050
$in this$	14	14	1759	0.008
$this way$	3	2.02	264	0.0077
$way \langle /s \rangle$	18	18	122	0.148

## 10.5 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.

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.

### 10.5.1 Linear Interpolation

Linear interpolation, also called deleted interpolation (Jelinek and Mercer 1980), combines linearly the maximum likelihood estimates from length 1 to  $n$ . For trigrams, 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 10.6 shows the interpolated probabilities of bigrams with  $\lambda_2 = 0.7$  and  $\lambda_1 = 0.3$ . The sentence probability using these interpolations is  $9.46 \times 10^{-45}$ .

**Table 10.6** 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

$w_{i-1}, w_i$	$C(w_{i-1}, w_i)$	$C(w_{i-1})$	$P_{\text{MLE}}(w_i   w_{i-1})$	$P_{\text{MLE}}(w_i)$	$P_{\text{Interp}}(w_i   w_{i-1})$
<code>&lt;s&gt; a</code>	133	7072	0.019	0.023	0.020
<i>a good</i>	14	2482	0.006	0.00049	0.0041
<i>good deal</i>	0	53	0.0	$4.62 \times 10^{-5}$	$1.38 \times 10^{-5}$
<i>deal of</i>	1	5	0.2	0.031	0.149
<i>of the</i>	742	3310	0.224	0.058	0.174
<i>the literature</i>	1	6248	0.00016	$6.47 \times 10^{-5}$	0.000131
<i>literature of</i>	3	7	0.429	0.031	0.309
<i>of the</i>	742	3310	0.224	0.058	0.174
<i>the past</i>	70	6248	0.011	0.00092	0.00812
<i>past was</i>	4	99	0.040	0.020	0.0344
<i>was indeed</i>	0	2211	0.0	0.00016	$4.71 \times 10^{-5}$
<i>indeed already</i>	0	17	0.0	0.00059	0.000177
<i>already being</i>	0	64	0.0	0.00074	0.000222
<i>being transformed</i>	0	80	0.0	$9.25 \times 10^{-6}$	$2.77 \times 10^{-6}$
<i>transformed in</i>	0	1	0.0	0.016	0.00488
<i>in this</i>	14	1759	0.008	0.0024	0.0063
<i>this way</i>	3	264	0.011	0.0011	0.00829
<i>way &lt;/s&gt;</i>	18	122	0.148	0.065	0.123

We can now understand why bigram *we the* is ranked so high in Table 10.1 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  $\lambda_1 P_{\text{MLE}}(\text{the})$  boosted the bigram to the top because of the high frequency of *the*.

### 10.5.2 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 unigrams. For a bigram language model, the back-off probability can be expressed as:

$$P_{\text{Backoff}}(w_i | w_{i-1}) = \begin{cases} P(w_i | w_{i-1}), & \text{if } C(w_{i-1}, w_i) \neq 0, \\ \alpha P(w_i), & \text{otherwise.} \end{cases}$$

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:

$$P_{\text{Backoff}}(w_i | w_{i-1}) = \begin{cases} P_{\text{MLE}}(w_i | w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}, & \text{if } C(w_{i-1}, w_i) \neq 0, \\ P_{\text{MLE}}(w_i) = \frac{C(w_i)}{\#\text{words}}, & \text{otherwise.} \end{cases}$$

and Table 10.7 shows the probability estimates we can derive from our small corpus. They yield a sentence probability of  $2.11 \times 10^{-40}$  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:

$$\begin{aligned} & P_{\text{Backoff}}(w_i | w_{i-k}, \dots, w_{i-1}) \\ &= \begin{cases} P_{\text{MLE}}(w_i | w_{i-k}, \dots, w_{i-1}), & \text{if } C(w_{i-k}, \dots, w_i) \neq 0, \\ \alpha P_{\text{Backoff}}(w_i | w_{i-k+1}, \dots, w_{i-1}), & \text{otherwise.} \end{cases} \end{aligned}$$

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.

**Table 10.7** Probability estimates using an elementary backoff technique

$w_{i-1}, w_i$	$C(w_{i-1}, w_i)$		$C(w_i)$	$P_{\text{Backoff}}(w_i   w_{i-1})$
<s>			7072	—
<s> <i>a</i>	133		2482	0.019
<i>a good</i>	14		53	0.006
<i>good deal</i>	0	Backoff	5	$4.62 \times 10^{-5}$
<i>deal of</i>	1		3310	0.2
<i>of the</i>	742		6248	0.224
<i>the literature</i>	1		7	0.00016
<i>literature of</i>	3		3310	0.429
<i>of the</i>	742		6248	0.224
<i>the past</i>	70		99	0.011
<i>past was</i>	4		2211	0.040
<i>was indeed</i>	0	Backoff	17	0.00016
<i>indeed already</i>	0	Backoff	64	0.00059
<i>already being</i>	0	Backoff	80	0.00074
<i>being transformed</i>	0	Backoff	1	$9.25 \times 10^{-6}$
<i>transformed in</i>	0	Backoff	1759	0.016
<i>in this</i>	14		264	0.008
<i>this way</i>	3		122	0.011
<i>way &lt;/s&gt;</i>	18		7072	0.148

### 10.5.3 Katz's Back-Off Model

As with linear interpolation in Sect. 10.5.1, 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 10.4 and 10.5 for our sentence. We then assign the remaining probability mass to the unigrams.

To compute  $\alpha$ , 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:

$$\sum_{w_i} P_{\text{Katz}}(w_i | w_{i-1}) = \sum_{w_i, C(w_{i-1}, w_i) > 0} \tilde{P}(w_i | w_{i-1}) + \alpha \sum_{w_i, C(w_{i-1}, w_i) = 0} P_{\text{MLE}}(w_i), \\ = 1.$$

We know that this sum equals 1, and we derive  $\alpha$  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:

$$P_{\text{Katz}}(w_i | w_{i-2}, w_{i-1}) \\ = \begin{cases} \tilde{P}(w_i | w_{i-2}, w_{i-1}), & \text{if } C(w_{i-2}, w_{i-1}, w_i) \neq 0, \\ \alpha(w_{i-2}, w_{i-1}) P_{\text{Katz}}(w_i | w_{i-1}), & \text{otherwise.} \end{cases}$$

### 10.5.4 Kneser–Ney Smoothing Model

The Kneser–Ney model (Ney et al. 1994) is our last method to smooth probability estimates.  $P_{KN}(w_i | w_{i-1})$  consists of two terms:

1. The first one uses the maximum likelihood to estimate the bigram probabilities when they exist minus a discount to make room for bigrams unseen in the training corpus. We ensure that the first term is always positive by taking the maximum of the discounted value and 0:

$$\frac{\max(0, C(w_{i-1}, w_i) - \delta)}{C(w_{i-1})} \text{ with } 0 < \delta < 1.$$

2. The second term,  $P_{KN}(w_i)$ , measures the percentage of unique bigrams with the second word in it with respect to the number of unique bigrams. We compute it from the set of bigrams seen at least once in the corpus

$$\{(w_k, w_{k+1}) : C(w_k, w_{k+1}) > 0\}$$

and we extract a subset of it, where the second word is  $w_i$ :  $\{(w_k, w_i)\}$ .  $P_{KN}(w_i)$  is defined as the ratio of their cardinalities:

$$P_{KN}(w_i) = \frac{|\{(w_k, w_i) : C(w_k, w_i) > 0\}|}{|\{(w_k, w_{k+1}) : C(w_k, w_{k+1}) > 0\}|}.$$

We scale this second term with  $\lambda_{w_{i-1}}$ .

Finally, we have:

$$P_{KN}(w_i | w_{i-1}) = \frac{\max(0, C(w_{i-1}, w_i) - \delta)}{C(w_{i-1})} + \lambda_{w_{i-1}} P_{KN}(w_i).$$

The absolute discounting value  $\delta$  is a constant, for instance 0.1, that we can optimize using a validation set. We must compute the  $\lambda_{w_{i-1}}$  values for all the words so that the  $P_{KN}(w_i | w_{i-1})$  probabilities have a sum of one.

The Kneser–Ney smoothing model has shown the best performances in language modeling especially for small corpora. See Sect. 10.7 on evaluation.

## 10.6 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.

## 10.7 Quality of a Language Model

### 10.7.1 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:

$$H(L) = -\frac{1}{n} \log_2 P(w_1, \dots, w_n).$$

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.

### 10.7.2 Entropy Rate

We used entropy with characters in Chap. 4, *Encoding and Annotation Schemes*. 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**:

$$H(L) = -\frac{1}{n} \sum_{w_1, \dots, w_n \in L} P(w_1, \dots, w_n) \log_2 P(w_1, \dots, w_n),$$

where  $L$  is the set of all possible sequences of length  $n$ .

It has been proven that when  $n \rightarrow \infty$  or  $n$  is very large and under certain conditions, we have

$$\begin{aligned} H(L) &= \lim_{n \rightarrow \infty} -\frac{1}{n} \sum_{w_1, \dots, w_n \in L} P(w_1, \dots, w_n) \log_2 P(w_1, \dots, w_n), \\ &= \lim_{n \rightarrow \infty} -\frac{1}{n} \log_2 P(w_1, \dots, w_n), \end{aligned}$$

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.

### 10.7.3 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. 6, *Topics in Information Theory and Machine Learning*, the cross entropy of  $M$  on  $P$  is given by:

$$H(P, M) = -\frac{1}{n} \sum_{w_1, \dots, w_n \in L} P(w_1, \dots, w_n) \log_2 M(w_1, \dots, w_n).$$

As for the entropy rate, it has been proven that, under certain conditions

$$\begin{aligned} H(P, M) &= \lim_{n \rightarrow \infty} -\frac{1}{n} \sum_{w_1, \dots, w_n \in L} P(w_1, \dots, w_n) \log_2 M(w_1, \dots, w_n), \\ &= \lim_{n \rightarrow \infty} -\frac{1}{n} \log_2 M(w_1, \dots, w_n). \end{aligned}$$

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. 6, *Topics in Information Theory and Machine Learning*, 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. 10.7.1.

### 10.7.4 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.

## 10.8 Generating Text from a Language Model

In early speech recognition systems, given a sequence of words,  $x_1, x_2, \dots, x_{i-1}$ , a vocabulary  $V$ , and an acoustic input,  $A$ , language models helped predict the next

word,  $x_i$ :

$$\arg \max_{x_i \in V} P(x_i | x_1, x_2, \dots, x_{i-1}, A),$$

so that we have the most likely sequence.

In this application, the prediction is conditioned by  $A$ . We can set it aside and only use the  $n$ -gram language model we derived from a corpus to predict the next word:

$$P(x_i | x_1, x_2, \dots, x_{i-1}).$$

By adding the predicted word to the existing sequence and repeating this operation, we will be able generate text:

$$\begin{aligned} & P(x_{i+1} | x_1, x_2, \dots, x_{i-1}, x_i), \\ & P(x_{i+2} | x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}), \\ & P(x_{i+3} | x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, x_{i+2}), \\ & \dots \end{aligned}$$

This will be pretty dull nonetheless. Given a starting word, possibly the start of sentence symbol `<s>`, and selecting the next word with the highest probability,

$$\arg \max_{x_i \in V} P(x_i | x_1, x_2, \dots, x_{i-1}),$$

we will always generate the same sequence ...

### 10.8.1 Using the Multinomial Distribution

Another option is to select the next word following the multinomial distribution of our language model. That is, for example, if a word  $w$  has three observed followers in the corpus,  $w_a$ ,  $w_b$ , and  $w_c$  with  $P(w_a|w) = 0.5$ ,  $P(w_b|w) = 0.3$ , and  $P(w_c|w) = 0.2$ , we will select  $w_a$  50% of the time,  $w_b$  30%, and  $w_c$  20%. We will then generate text with the same statistical properties as our training corpus.

To select an outcome from a `distribution` in the form of a list of real values, for instance [0.5, 0.3, 0.2], we can use:

```
np.random.multinomial(1, distribution)
```

which will return a one-hot vector: [1, 0, 0], 50% of the time, [0, 1, 0], 30%, and [0, 0, 1], 20%. We can then extract the word index with `np.argmax()`:

```
np.argmax(np.random.multinomial(1, distribution))
```

Let us take an example with the *Iliad* corpus set in lowercase and use bigrams to simplify:

$$P(x_i | x_{i-1})$$

We first tokenize the text that we store in the `words` list. We count the unigrams, bigrams, and we compute the conditional probabilities with:

```
unigram_freqs = Counter(words)

bigrams = [tuple(words[idx:idx + 2])
           for idx in range(len(words) - 1)]
bigram_freqs = Counter(bigrams)

cond_probs = {k: v/unigram_freqs[k[0]] for
              k, v in bigram_freqs.items()}
```

Starting from the last word in the sequence, say *Hector*, we estimate

$$P(x | \text{hector})$$

from the corpus. For a given word, we extract the conditional probabilities with:

```
def bigram_dist(word, cond_probs):
    bigram_cprobs = sorted(
        [(k, v) for k, v in cond_probs.items()
         if k[0] == word],
        key=lambda tup: tup[1], reverse=True)
    return bigram_cprobs
```

We have 184 bigrams (*hector*, *x*), for which the five highest probabilities are:

$$\begin{aligned} P(\text{and}|\text{hector}) &= 0.117, \\ P(\text{son}|\text{hector}) &= 0.052, \\ P(\text{s}|\text{hector}) &= 0.048, \\ P(\text{was}|\text{hector}) &= 0.031, \\ P(\text{in}|\text{hector}) &= 0.023. \end{aligned}$$

The generation is then easy. We start with a word, here *Hector*. We then run a loop, where we select the next word according to the multinomial distribution and replace the current word with the next word:

```
print(start_word, end=' ')
current_word = start_word
for i in range(50):
    bigram_cprobs = bigram_dist(current_word, cond_probs)
    dist = [bigram_cprob[1] for bigram_cprob in bigram_cprobs]
    selected_idx = np.argmax(np.random.multinomial(1, dist))
    next_bigram_cprob = bigram_cprobs[selected_idx]
    current_word = next_bigram_cprob[0][1]
    print(current_word, end=' ')
```

This loop creates a text that will change each time we run it:

hector shouted to rush without a brave show but flow for his tail from their cave even for he killed in despair as luck will override fate of all gods who were thus beaten wild in all good was dead in spite him and called you are freshly come close to ...

### 10.8.2 Transforming the Distribution

In the previous section, we generated a text with a bigram distribution that is the same as that of the training corpus i.e. the *Iliad*. We can transform it to make it more deterministic or more random.

Chollet (2021, pp. 369–373) proposed a transformation over the second word of the bigram using a power function that has this property. He called the inverse of the power, the *temperature*,  $T$ , referring to the temperature in Boltzmann distribution:

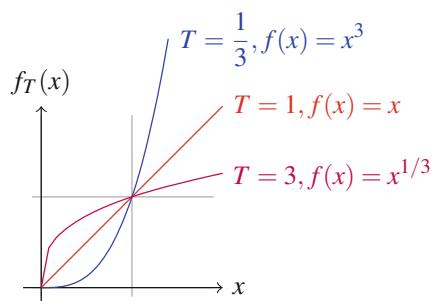
$$f_T(x) = x^{\frac{1}{T}}.$$

As input, we take the original bigram distribution, transform it, and normalize it so that the probabilities sum to 1.

```
def power_transform(distribution, T=0.5):
    new_dist = np.power(distribution, 1/T)
    return new_dist / np.sum(new_dist)
```

Figure 10.1 shows the power function with different temperatures. Given that the probabilities range from 0 to 1, a temperature of 1 yields the original distribution; a low temperature squashes low probabilities and accentuates higher probabilities. This makes the predictions more deterministic. On the contrary, a high temperature equalizes the distribution and makes the predictions more random.

**Fig. 10.1** The power function between 0 and 1 with the exponents  $\frac{1}{3}$ , 1, and 3



## 10.9 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.

### 10.9.1 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).<sup>1</sup> Mutual

---

<sup>1</sup> 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 10.8** Collocates of *surgery* extracted from the Bank of English using the mutual information test. Note the misspelled word *pioneer*

Word	Frequency	Bigram word + <i>surgery</i>	Mutual information
<i>Arthroscopic</i>	3	3	11.822
<i>Pioneer</i>	3	3	11.822
<i>Reconstructive</i>	14	11	11.474
<i>Refractive</i>	6	4	11.237
<i>Rhinoplasty</i>	5	3	11.085

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 10.8 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.

### 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  $P(w_i, w_j) = P(w_i) \times P(w_j)$ .

**Table 10.9** Collocates of *set* extracted from Bank of English using the *t*-score

Word	Frequency	Bigram <i>set</i> + word	<i>t</i> -score
<i>up</i>	134, 882	5512	67.980
<i>a</i>	1, 228, 514	7296	35.839
<i>to</i>	1, 375, 856	7688	33.592
<i>off</i>	52, 036	888	23.780
<i>out</i>	12, 3831	1252	23.320

The *t*-score computes the difference between the two hypotheses, respectively,  $\text{mean}(P(w_i, w_j))$  and  $\text{mean}(P(w_i))\text{mean}(P(w_j))$ , and divides it by the variances. It is defined by the formula:

$$t(w_i, w_j) = \frac{\text{mean}(P(w_i, w_j)) - \text{mean}(P(w_i))\text{mean}(P(w_j))}{\sqrt{\sigma^2(P(w_i, w_j)) + \sigma^2(P(w_i))P(w_j))}}.$$

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 10.9 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  $\log f(k; N, p) = k \log p + (N - k) \log(1 - p)$ .

**Table 10.10** 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

	$w_i$	$\neg w_i$
$w_j$	$C(w_i, w_j)$	$C(\neg w_i, w_j) = C(w_j) - C(w_i, w_j)$
$\neg w_j$	$C(w_i, \neg w_j) = C(w_i) - C(w_i, w_j)$	$C(\neg w_i, \neg w_j) = N - C(w_i, w_j)$

Using the counts in Table 10.10 and the maximum likelihood estimate, we have

$$\begin{aligned} p &= P(w_j) = \frac{C(w_j)}{N}, \\ p_1 &= P(w_j|w_i) = \frac{C(w_i, w_j)}{C(w_i)}, \text{ and} \\ p_2 &= P(w_j|\neg w_i) = \frac{C(w_j) - C(w_i, w_j)}{N - C(w_i)}, \end{aligned}$$

where  $N$  is the number of words in the corpus.

### 10.9.2 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. 9.4.2 and 10.2.1. 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 information 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:

```
words = tokenize(text.lower())
```

```

word_freqs = Counter(words)
bigrams = [tuple(words[idx:idx + 2])
           for idx in range(len(words) - 1)]
bigram_freqs = Counter(bigrams)
mi = mutual_info(words, word_freqs, bigram_freqs)

for bigram in sorted(mi.keys(), key=lambda x: (-mi.get(x), x)):
    print(bigram, '\t',
          word_freqs[bigram[0]], '\t',
          word_freqs[bigram[1]], '\t',
          bigram_freqs[bigram], '\t',
          mi[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, word_freqs, bigram_freqs):
    ts = {}
    for bigram in bigram_freqs:
        ts[bigram] = ((bigram_freqs[bigram] -
                      word_freqs[bigram[0]] *
                      word_freqs[bigram[1]] /
                      len(words)) /
                      math.sqrt(bigram_freqs[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, word_freqs, bigram_freqs):
    lr = {}
    for bigram in bigram_freqs:
        p = word_freqs[bigram[1]] / len(words)
        p1 = bigram_freqs[bigram] / word_freqs[bigram[0]]
        p2 = ((word_freqs[bigram[1]] - bigram_freqs[bigram]) /
               (len(words) - word_freqs[bigram[0]]))
        if p1 != 1.0 and p2 != 0.0:
            lr[bigram] = 2.0 * (
                log_f(bigram_freqs[bigram],

```

```

        word_freqs[bigram[0]], p1) +
log_f(word_freqs[bigram[1]] -
      bigram_freqs[bigram],
      len(words) - word_freqs[bigram[0]], p2) -
log_f(bigram_freqs[bigram],
      word_freqs[bigram[0]], p) -
log_f(word_freqs[bigram[1]] -
      bigram_freqs[bigram],
      len(words) - word_freqs[bigram[0]], p))

return lr

def log_f(k, N, p):
    return k * math.log(p) + (N - k) * math.log(1 - p)

```

### 10.9.3 Applying Collocation Measures

To observe concretely the collocations extracted by the three measures, we applied them to our Homer corpus consisting of *the Iliad* and *the Odyssey*. We set the text in lowercase, tokenized it, and ran the programs from Sect. 10.9.2. Table 10.11 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 10.11 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 10.11 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.

**Table 10.11** 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

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

## 10.10 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).

Jurafsky and Martin (2008) as well as Manning and Schütze (1999) are good references on language models in general. Chen and Goodman (1998) give additional details on language modeling techniques and Dunning (1993) on  $\chi^2$  tests and likelihood ratios to improve collocation detection.

The Natural Language Toolkit (NLTK) provides modules for  $n$ -grams computation, collocations as well as language models. The KenLM Language Model Toolkit (Heafield 2011) is a fast implementation of the Kneser-Ney algorithm. See also Norvig (2009) for beautiful Python programs building on language models to segment words, decipher codes, or check word spelling.

# Chapter 11

## Dense Vector Representations



In Sect. 6.3, we used one-hot encoding to represent words and, in Sect. 9.5.3, bags of words to represent documents. When applied to any significant corpus, both techniques often result in very large, sparse matrices, i.e. containing many zeros. In this chapter, we will examine how we can build **dense vector representations** ranging from two to a few hundred dimensions instead. In the context of natural language processing, we will call **embeddings** these dense vectors.

### 11.1 Vector Representations

As first 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. 7.1, we already used the counts of letter *A* to discriminate English and French. Figure 11.1 shows the rest of the counts for all the characters, set to lower case, broken down by chapter; 30 in total.

As shown by this table, we can well represent the chapters (the documents) by vectors of characters counts, instead of words. We could call this representation a bag of characters. In Figure 11.1, there are as many as 40 characters: the 26 unaccented letters from *a* to *z* and 14 French accented letters: à, â, æ, ç, è, é, ê, ë, ï, ï, ô, œ, ù, and û. This means that we represent each of our 30 chapters by a vector of 40 dimensions, where the chapter coordinates are the counts.

Conversely, we can represent the characters by their counts in the chapters. We just need to store Fig. 11.1 in a  $(30 \times 40)$  matrix and transpose it. This results in the  $(40 \times 30)$  matrix shown in Fig. 11.2, where the 40 rows are the vector representations of each character in 30 dimensions: The chapters.

Ch.	a	b	c	d	e	...	v	w	x	y	z	à	â	æ	ç	è	é	...	û	œ
01_fr	2503	365	857	1151	4312	...	414	0	129	94	20	128	36	0	35	102	423	...	7	5
02_fr	2992	391	1006	1388	4993	...	499	0	175	89	23	136	50	1	28	147	513	...	9	5
03_fr	1042	152	326	489	1785	...	147	0	42	31	7	39	9	0	10	49	194	...	7	2
04_fr	2487	303	864	1137	4158	...	422	0	138	81	27	110	43	0	22	138	424	...	4	8
05_fr	2014	268	645	949	3394	...	315	1	83	67	18	90	67	0	24	112	367	...	15	7
06_fr	2805	368	910	1266	4535	...	453	0	151	80	39	131	42	0	30	122	548	...	15	9
07_fr	5062	706	1770	2398	8512	...	844	0	272	148	71	246	50	1	46	232	966	...	38	9
08_fr	2643	325	869	1085	4229	...	437	0	135	64	30	130	43	0	34	119	502	...	8	5
09_fr	2126	289	771	920	3599	...	348	0	119	58	20	90	24	2	16	99	370	...	15	3
10_fr	1784	249	546	805	3002	...	270	0	65	61	11	73	18	0	16	68	304	...	10	5
11_fr	2641	381	817	1078	4306	...	425	0	114	61	25	101	40	0	34	108	438	...	9	7
12_fr	2766	373	935	1237	4618	...	455	0	149	98	37	129	33	0	23	151	480	...	14	0
13_fr	5047	725	1730	2273	8678	...	767	0	288	119	41	209	55	3	61	237	940	...	30	13
14_fr	5312	689	1754	2149	8870	...	914	0	283	145	41	224	75	0	56	260	1019	...	21	12
15_fr	1215	173	402	582	2195	...	206	0	63	36	3	48	20	2	17	58	221	...	11	6
01_en	2217	451	729	1316	3967	...	258	653	29	401	18	0	0	0	0	0	0	...	0	0
02_en	2761	551	777	1548	4543	...	295	769	37	475	31	0	0	0	0	0	0	...	0	0
03_en	990	183	271	557	1570	...	94	254	8	145	15	0	0	0	0	0	0	...	0	0
04_en	2274	454	736	1315	3814	...	245	663	60	467	19	0	0	0	0	0	0	...	0	0
05_en	1865	400	553	1135	3210	...	194	568	26	330	33	0	0	0	0	0	0	...	0	0
06_en	2606	518	797	1509	4237	...	277	733	49	464	37	0	0	0	0	0	0	...	0	0
07_en	4805	913	1521	2681	7834	...	465	1332	74	843	52	0	0	0	0	0	0	...	0	0
08_en	2396	431	702	1416	4014	...	266	695	65	379	24	0	0	0	0	0	0	...	0	0
09_en	1993	408	653	1096	3373	...	208	560	25	328	18	0	0	0	0	0	0	...	0	0
10_en	1627	359	451	933	2690	...	181	410	31	255	20	0	0	0	0	0	0	...	0	0
11_en	2375	437	643	1364	3790	...	246	632	20	457	39	0	0	0	0	0	0	...	0	0
12_en	2560	489	757	1566	4331	...	278	721	35	418	40	0	0	0	0	0	0	...	0	0
13_en	4597	987	1462	2689	7963	...	437	1374	77	673	49	0	0	0	0	0	0	...	0	0
14_en	4871	948	1439	2799	8179	...	539	1377	90	856	49	0	0	0	0	0	0	...	0	0
15_en	1119	229	335	683	1994	...	108	330	14	150	9	0	0	0	0	0	0	...	0	0

**Fig. 11.1** Character counts per chapter, where the fr and en suffixes designate the language, either French or English. In total, there are as many as 40 characters: the 26 unaccented letters from *a* to *z* and 14 French accented letters: à, â, é, etc.

## 11.2 Dimensionality Reduction

Should we want to represent any document in any language by character counts, we would potentially need the whole Unicode character set, where the code points range from 0 to U+10FFFF. This corresponds to a decimal value of 1,114,111. This then sets the dimensionality ceiling to more than 1 million, where, for one document, most of the values would certainly be zeros.

Fortunately, it is possible to reduce considerably this number using a technique called **singular value decomposition** (SVD) that will keep the resulting vectors semantically close to their original values.

### 11.2.1 Singular Value Decomposition

Let us apply this decomposition to the data in Fig. 11.1. Let us denote  $X$  the  $m \times n$  matrix of the letter counts per chapter, in our case,  $m = 30$  and  $n = 40$ . From the

	French										English									
	01	02	03	04	05	...	12	13	14	15	01	02	03	04	05	...	12	13	14	15
a	2503	2992	1042	2487	2014	...	2766	5047	5312	1215	2217	2761	990	2274	1865	...	2560	4597	4871	1119
b	365	391	152	303	268	...	373	725	689	173	451	551	183	454	400	...	489	987	948	229
c	857	1006	326	864	645	...	935	1730	1754	402	729	777	271	736	553	...	757	1462	1439	335
d	1151	1388	489	1137	949	...	1237	2273	2149	582	1316	1548	557	1315	1135	...	1566	2689	2799	683
e	4312	4993	1785	4158	3394	...	4618	8678	8870	2195	3967	4543	1570	3814	3210	...	4331	7963	8179	1994
f	264	319	136	314	223	...	329	648	628	150	596	685	279	595	515	...	677	1254	1335	323
g	349	360	122	331	215	...	350	566	630	134	662	769	253	559	525	...	650	1201	1140	281
h	295	350	126	287	242	...	349	642	673	148	2060	2530	875	1978	1693	...	2348	4278	4534	1108
i	1945	2345	784	2028	1617	...	2273	3940	4278	969	1823	2163	783	1835	1482	...	2033	3634	3829	912
j	65	81	41	57	67	...	65	140	143	27	22	13	4	22	7	...	28	39	36	9
k	4	6	7	3	3	...	2	22	2	6	200	284	82	198	153	...	234	432	427	112
l	1946	2128	816	1796	1513	...	1955	3746	3780	950	1204	1319	520	1073	949	...	1102	2281	2218	579
m	726	832	397	722	651	...	812	1597	1610	387	656	829	333	690	571	...	746	1493	1534	351
n	1896	2308	778	1958	1547	...	2285	3984	4255	906	1851	2218	816	1771	1468	...	2125	3774	4053	924
o	1372	1560	612	1318	1053	...	1419	2736	2713	697	1897	2237	828	1865	1586	...	2105	3911	3989	1004
p	789	977	315	773	672	...	865	1550	1599	417	525	606	194	514	517	...	581	1099	1019	305
q	248	281	102	274	166	...	272	425	512	103	19	21	13	33	17	...	32	49	36	9
r	1948	2376	792	2000	1601	...	2276	4081	4271	985	1764	2019	711	1726	1357	...	1939	3577	3689	863
s	2996	3454	1174	2792	2192	...	3131	5599	5770	1395	1942	2411	864	1918	1646	...	2152	3894	3946	997
t	1938	2411	856	2031	1736	...	2274	4387	4467	1037	2547	3083	1048	2704	2178	...	3046	5540	5858	1330
u	1792	2069	707	1734	1396	...	1923	3480	3697	893	704	861	298	745	663	...	750	1379	1490	310
v	414	499	147	422	315	...	455	767	914	206	258	295	94	245	194	...	278	437	539	108
w	0	0	0	0	1	...	0	0	0	0	653	769	254	663	568	...	721	1374	1377	330
x	129	175	42	138	83	...	149	288	283	63	29	37	8	60	26	...	35	77	90	14
y	94	89	31	81	67	...	98	119	145	36	401	475	145	467	330	...	418	673	856	150
z	20	23	7	27	18	...	37	41	41	3	18	31	15	19	33	...	40	49	49	9
à	128	136	39	110	90	...	129	209	224	48	0	0	0	0	0	...	0	0	0	0
â	36	50	9	43	67	...	33	55	75	20	0	0	0	0	0	...	0	0	0	0
æ	0	1	0	0	0	...	0	3	0	2	0	0	0	0	0	...	0	0	0	0
ç	35	28	10	22	24	...	23	61	56	17	0	0	0	0	0	...	0	0	0	0
è	102	147	49	138	112	...	151	237	260	58	0	0	0	0	0	...	0	0	0	0
é	423	513	194	424	367	...	480	940	1019	221	0	0	0	0	0	...	0	0	0	0
ê	43	68	24	36	44	...	60	126	94	32	0	0	0	0	0	...	0	0	0	0
ë	1	0	0	0	1	...	0	0	0	0	0	0	0	0	0	...	0	0	0	0
í	17	20	12	15	11	...	13	32	28	12	0	0	0	0	0	...	0	0	0	0
í	2	0	0	2	8	...	3	5	2	0	0	0	0	0	0	...	0	0	0	0
ô	20	20	27	15	23	...	15	37	45	24	0	0	0	0	0	...	0	0	0	0
ù	14	9	4	6	18	...	11	24	21	7	0	0	0	0	0	...	0	0	0	0
û	7	9	7	4	15	...	14	30	21	11	0	0	0	0	0	...	0	0	0	0
œ	5	5	2	8	7	...	0	13	12	6	0	0	0	0	0	...	0	0	0	0

**Fig. 11.2** Character counts per chapter in French, left part, and English, right part. In total, 30 chapters

works of Beltrami (1873), Jordan (1874), and Eckart and Young (1936), we know we can rewrite  $X$  as:

$$X = U \Sigma V^T,$$

where  $U$  is a matrix of dimensions  $m \times m$ ,  $\Sigma$ , a diagonal matrix of dimensions  $m \times n$ , and  $V$ , a matrix of dimensions  $n \times n$ . The diagonal terms of  $\Sigma$  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 Salammbo's chapters, have a maximal variance. The columns of  $V$  are called the principal axes and the columns of  $U\Sigma$ , the principal components. We have the properties:  $UUT^T = I$  and  $VV^T = I$ , where  $I$  is the identity matrix.

We use the product  $U\Sigma$  to reduce the initial dimension of the rows,  $n$ , by setting some values of  $\Sigma$  to zero, starting from the lowest ones. To reduce the dimensionality to 2, we only keep to two largest values of  $\Sigma$  and it results in matrices of size  $m \times 2$ . If we keep three values, our matrices will be of size  $m \times 3$ , etc.

### 11.2.2 Data Representation and Preprocessing

The NumPy, PyTorch, and scikit-learn libraries have SVD functions that we can call without bothering about the mathematical details. To store  $X$ , depending on the toolkit, we use a NumPy array as in Sect. 6.3 or a PyTorch tensor. Prior to the decomposition, we usually 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. 8.5.1, it is sometimes beneficial to apply a normalization before the standardization as in the next statements:

```
from sklearn.preprocessing import StandardScaler, Normalizer

X_norm = Normalizer().fit_transform(X)
X_scaled = StandardScaler().fit_transform(X_norm)
```

### 11.2.3 Computing a Singular Value Decomposition

To carry out the singular value decomposition, 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  $\Sigma$ . We do not need to compute the full squared  $U$  matrix as the values of  $\Sigma$  outside its diagonal are zero and we set `full_matrices=False`. We compute the new coordinates of the chapters, corresponding to the rows of  $Us$ , by multiplying  $U$  and the diagonal matrix of the singular values, `np.diag(s)`.

## 11.3 Applying a SVD to the *Salammbô* Dataset

We will now apply the dimensionality reduction the data in Fig. 11.1 and we will proceed in two steps:

1. We will first restrict the dataset to the French chapters and to two rows: The frequencies of letter  $A$  by chapter and the total counts of characters, ( $\#A + \#B + \#C + \dots$ ), as shown in Table 7.1. This small set has only two dimensions and is then easy to visualize;
2. We will then proceed with the counts of each character in Fig. 11.1 to have a complete example.

### 11.3.1 Counts of Letter A

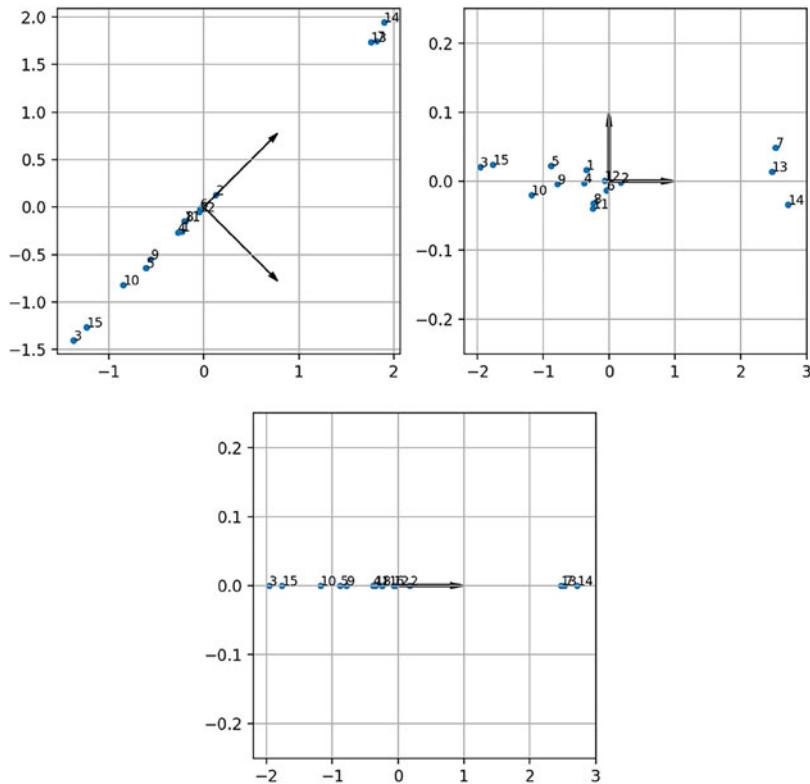
For the first experiment, we store the data in Table 7.1, left part, in a matrix,  $X_{\text{original}}$ , and standardize it,  $X_{\text{scaled}}$ . With this very small dataset, we set aside the normalization.

$$X_{\text{original}} = \begin{bmatrix} 36961. & 2503. \\ 43621. & 2992. \\ 15694. & 1042. \\ 36231. & 2487. \\ 29945. & 2014. \\ 40588. & 2805. \\ 75255. & 5062. \\ 37709. & 2643. \\ 30899. & 2126. \\ 25486. & 1784. \\ 37497. & 2641. \\ 40398. & 2766. \\ 74105. & 5047. \\ 76725. & 5312. \\ 18317. & 1215. \end{bmatrix}; X_{\text{scaled}} = \begin{bmatrix} -0.232 & -0.2556 \\ 0.1245 & 0.1275 \\ -1.3706 & -1.4001 \\ -0.2711 & -0.2681 \\ -0.6076 & -0.6386 \\ -0.0379 & -0.019 \\ 1.818 & 1.749 \\ -0.192 & -0.1459 \\ -0.5566 & -0.5509 \\ -0.8464 & -0.8188 \\ -0.2033 & -0.1475 \\ -0.048 & -0.0496 \\ 1.7565 & 1.7373 \\ 1.8967 & 1.9449 \\ -1.2302 & -1.2645 \end{bmatrix}$$

We then apply a SVD to the standardized matrix. The decomposition returns:

$$U = \begin{bmatrix} -0.0630 & 0.1783 \\ 0.0325 & -0.0225 \\ -0.3577 & 0.2233 \\ -0.0696 & -0.0228 \\ -0.1609 & 0.2348 \\ -0.0073 & -0.1429 \\ 0.4606 & 0.5228 \\ -0.0436 & -0.3491 \\ -0.1430 & -0.0429 \\ -0.2150 & -0.2087 \\ -0.0453 & -0.4232 \\ -0.0126 & 0.0115 \\ 0.4511 & 0.1455 \\ 0.4960 & -0.3645 \\ -0.3221 & 0.2605 \end{bmatrix}; \Sigma = \begin{bmatrix} 5.4764 & 0 \\ 0 & 0.0933 \end{bmatrix}; V = \begin{bmatrix} 0.7071 & 0.7071 \\ 0.7071 & -0.7071 \end{bmatrix}.$$

The first column vector in  $V$  corresponds to the direction of the regression line given in Sect. 7.3, while the second one is orthogonal to it;  $\Sigma$  contains the singular



**Fig. 11.3** Left pane: The standardized dataset with the singular vectors; Middle pane: The rotated dataset, note the change of scale; Right pane: The projection on the first (horizontal) singular vector

values; and  $U \Sigma$  contains the coordinates of the chapters in the new system defined by  $V$ . We project the points on an axis, for instance the first one, by simply setting all the other values of  $\Sigma$  to 0, here the second one:

$$\Sigma = \begin{bmatrix} 5.4764 & 0 \\ 0 & 0 \end{bmatrix}.$$

Figure 11.3 shows the results of the analysis on three panes. The left pane shows the original standardized dataset; the middle pane, the rotated dataset, corresponding to  $U \Sigma$ , and the right pane, the projection on the first singular vector. This corresponds to a reduction from 2 to 1 and is equivalent to a projection on the regression line.

### 11.3.2 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 Fig. 11.1. This decomposition results in a vector of 30 singular values ranging from 30.17 to  $1.34 \cdot 10^{-14}$  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 dimensionality from 30 to 2, while keeping a good deal of the variance between the points. To carry this out, we just set all the singular values in  $\Sigma$  to 0 except the two first ones. Figure 11.4 shows the projected chapters corresponding to the rows in the truncated  $U\Sigma$  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 on the  $x$  axis, corresponding to the English and French versions. In addition, although far from perfect, we can apply a translation with a vector that we could call `vFrench_to_English` to predict the approximate position of an English chapter from the position of the French chapter:

$$\text{chapter\_in\_English} \approx \text{chapter\_in\_French} + \text{vFrench\_to\_English}.$$

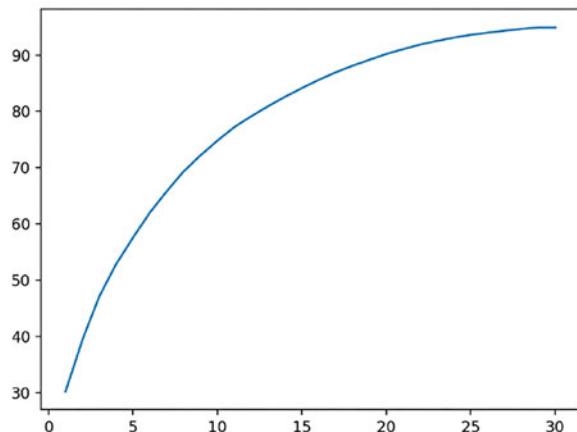
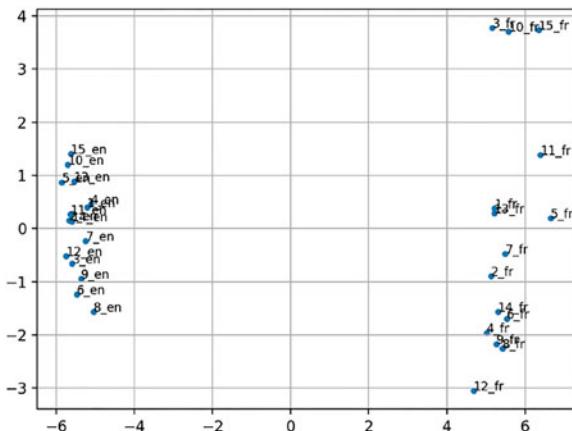
On the right pane of the figure, we have the cumulative sum of the singular values: 39.37 for the two first ones and 94.90 for all.

### 11.3.3 The Characters in a Space of Documents

In Fig. 11.1, 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  $X$ , apply the decomposition, and truncate  $U\Sigma$ .

Figure 11.5 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 corner; letters rare in French and English like Ê or Æ are in the left corner; letters that are rare in French, but frequent in English are on the right; while letters 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 rarity–frequency axis.

**Fig. 11.4** 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 make up about 40% of the total

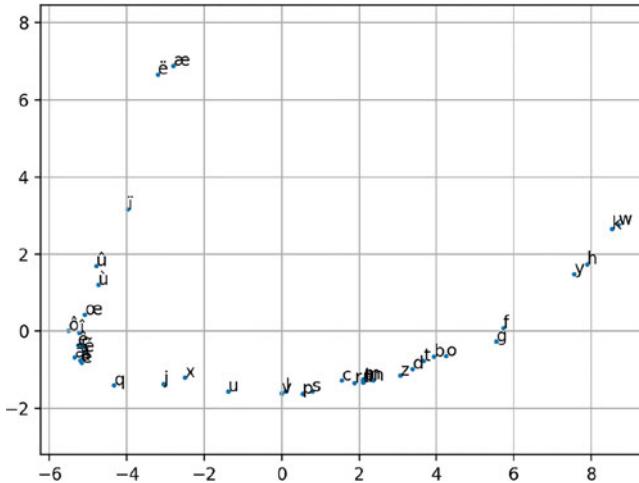


### 11.3.4 Singular Value Decomposition and Principal Component Analysis

In the NLP literature, we find two names for the dimensionality reduction technique we have just seen: SVD and principal component analysis (PCA). They often mean the same thing though there are a few differences. PCA is a sort of ready-to-use application of SVD:

- It usually includes a standardization, or at least a centering;
- the output dimensionality is a parameter of the program; and
- it only returns the  $U \Sigma$  or  $U$  matrices.

We used NumPy to compute the singular value decompositions. PyTorch has a similar function: `torch.linalg.svd`. scikit-learn provides two classes for this, `PCA` and `TruncatedSVD`. Both programs have an argument to set the number of dimensions



**Fig. 11.5** Characters projected on a plane

to keep and return a truncated matrix. The difference between them is that `PCA` centers the columns, while `TruncatedSVD` applies to raw matrices. Finally, `fbpca` is Facebook's very fast implementation of PCA. By default, it also centers the columns.

## 11.4 Latent Semantic Indexing

We can extend the technique we described for characters in the previous section to words. The matrix structure will be the same, but the rows will correspond to the words in the corpus, and the columns, to documents, or more simply a context of a few words, for instance a paragraph. The matrix of word-document pairs ( $w_i, D_j$ ) in Table 11.1 is just the transpose of Table 9.8. Each matrix element measures the association strength between word  $w_i$  and document  $D_j$ .

In Fig. 11.1, the matrix elements are simply the raw counts of the pairs. Deerwester et al. (1990) used a more elaborate formula similar to  $tf \times idf$ , defined in Sect. 9.5.3. It consists of the product 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  $idf$ .

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

$$1 - \sum_j \frac{p_{ij} \log(p_{ij})}{\log(n\text{docs})},$$

**Table 11.1** The word-by-document matrix. Each cell  $(w_i, D_j)$  contains an association score between  $w_i$  and  $D_j$

Words\D#	$D_1$	$D_2$	$D_3$	...	$D_n$
$w_1$	$X(w_1, D_1)$	$X(w_1, D_2)$	$X(w_1, D_3)$	...	$X(w_1, D_n)$
$w_2$	$X(w_2, D_1)$	$X(w_2, D_2)$	$X(w_2, D_3)$	...	$X(w_2, D_n)$
$w_3$	$X(w_3, D_1)$	$X(w_3, D_2)$	$X(w_3, D_3)$	...	$X(w_3, D_n)$
...	...	...	...	...	...
$w_m$	$X(w_m, D_1)$	$X(w_m, D_2)$	$X(w_m, D_3)$	...	$X(w_m, D_n)$

where  $p_{ij} = \frac{tf_{ij}}{gf_i}$ ,  $gf_i$  is the global frequency of term  $i$ : The total number of times it occurs in the collection, and  $ndocs$ , the total number of documents.

Once we have filled the matrix, we can apply a SVD to reduce the dimensionality and represent the words or transpose the matrix before and represent the documents. Deerwester et al. (1990) used this method to index the documents of a collection, where they reduced the dimension of the vector space to 100, i.e. each document is represented by a 100-dimensional vector. They called it **latent semantic indexing** (LSI). We can store the resulting documents in a vector database that will enable us to speed up document queries and comparisons.

Similarly to LSI, Benzécri (1981a) and Benzécri and Morfin (1981) defined a correspondence analysis method that applies the  $\chi^2$  metric to pairs of words or word-document pairs and a specific normalization of the matrices.

## 11.5 Word Embeddings from a Cooccurrence Matrix

Latent semantic indexing uses documents as the context of a word. Nonetheless, in many cases, we have less documents than words and this makes it difficult to use this technique to compute word embeddings. Imagine a corpus of one single very long document for instance. Instead of documents, we can consider a window to the left and to the right of a focus word:  $w_i$ . The word context  $C_j$ , replacing document  $D_j$ , is then defined by a window of  $2K$  words centered on  $w_i$ :

$$w_{i-K}, w_{i-K+1}, \dots, w_{i-1}, \underline{w_i}, w_{i+1}, \dots, w_{i+K-1}, w_{i+K},$$

Using this context, a very simple replacement of the association scores  $X(w_i, D_j)$  in Table 11.1 is to use the counts:  $C_{i \neq k}(w_i, w_k)$ ,  $w_k \in C_j$ , computed over all the contexts of  $w_i$  in the corpus as in Table 11.2. The size of this matrix is  $n \times n$ , where  $n$  is the number of unique words in the corpus. A PCA will enable us to truncate the columns of  $U\Sigma$  to 50, 100, 300, or 500 dimensions and create word embeddings from its rows.

After this description of cooccurrence matrices, we will now see how to build them from a corpus and reduce the dimensionality of their rows with a PCA.

**Table 11.2** The word-by-context matrix. Counts of bigrams  $(w_i, w_j)$ , where  $w_j$  occurs in a window of size  $2K$  centered on  $w_i$  called the context

Words\C#	$w_1$	$w_2$	$w_3$	...	$w_n$
$w_1$	$C(w_1, w_1)$	$C(w_1, w_2)$	$C(w_1, w_3)$	...	$C(w_1, w_n)$
$w_2$	$C(w_2, w_1)$	$C(w_2, w_2)$	$C(w_2, w_3)$	...	$C(w_2, w_n)$
$w_3$	$C(w_3, w_1)$	$C(w_3, w_2)$	$C(w_3, w_3)$	...	$C(w_3, w_n)$
...	...	...	...	...	...
$w_n$	$C(w_n, w_1)$	$C(w_n, w_2)$	$C(w_n, w_3)$	...	$C(w_n, w_n)$

### 11.5.1 Preprocessing

Before we can build the matrix, we need to preprocess our corpus: We tokenize it with:

```
text = open('corpus.txt', encoding='utf8').read()
text = text.lower()
words = re.findall('\p{L}+', text)
```

We index the words:

```
vocab = sorted(list(set(words)))
vocab_size = len(vocab)
idx2word = dict(enumerate(vocab))
word2idx = {v: k for k, v in idx2word.items()}
```

We then replace the tokens in the list with their numerical index:

```
words_idx = [word2idx[word] for word in words]
```

### 11.5.2 Counting the Cooccurrences

Once we have a list of word indices, we can build the cooccurrence matrix, where for each word in the corpus, we will count the words in its context. These counts quantify the association between two words. However, a word that is 10 words apart from the focus is probably less significant than an adjacent one. To take this into account, we will weight the contribution of each context word by  $1/d$ ,  $d$  being the distance to the focus word. For each word, we store the current weighted counts in a dictionary and we update them with those of new left and right contexts with the `update_counts()` function.

```
def update_counts(start_dict: dict[int, float],
                  lc: list[int],
                  rc: list[int]) -> dict[int, float]:
    for i, word in enumerate(rc, start=1):
        if word in start_dict:
            start_dict[word] += 1.0/i
        else:
```

```

        start_dict[word] = 1.0/i
    for i, word in enumerate(lc[::-1], start=1):
        if word in start_dict:
            start_dict[word] += 1.0/i
        else:
            start_dict[word] = 1.0/i
    return start_dict

```

We store all the cooccurrence counts in a dictionary, `C_dict`, where the keys are the unique words of the corpus represented by their indices. For a given key, the value is another dictionary with all the words found in its contexts and their weighted counts. To compute the counts, we traverse the list of indices representing the corpus, where at each index, if the key is not already present, we create a dictionary; we extract the left and right contexts; and we update this dictionary with the weighted counts of the contexts.

```

def build_C(words_idx: list[int],
            K: int) -> dict[int, dict[int, float]]:
    C_dict = dict()
    for i, word in tqdm(enumerate(words_idx)):
        if word not in C_dict:
            C_dict[word] = dict()
        lc = words_idx[i - K:i]
        rc = words_idx[i + 1:i + K + 1]
        C_dict[word] = update_counts(C_dict[word],
                                      lc, rc)
    return C_dict

```

We call this function with:

```
C_dict = build_C(words_idx, K)
```

where  $K$  is the context size, for instance 4.

### 11.5.3 Applying a PCA

We have now the data needed in Table 11.2 and we can reduce its dimensionality. Before that, as our numerical libraries we can only apply a PCA to a matrix, we must convert our cooccurrence dictionary to a NumPy array. For this, we create a square matrix of zeros of the size of the vocabulary and we assign it the values in the dictionary:

```

def build_matrix(C_dict):
    cooc_mat = np.zeros((len(C_dict), len(C_dict)))
    for k, coocs in C_dict.items():
        for c, cnt in coocs.items():
            cooc_mat[k, c] = float(cnt)
    return cooc_mat

```

We call this function with:

```
cooc_mat = build_matrix(C_dict)
```

We compute the  $U$ ,  $\Sigma$ , and  $V$  matrices with Facebook's PCA as it is very fast. In the next statement, we set the reduced dimension of  $U$ 's row vectors to 50:

```
(U, s, Va) = fbpc.pca(cooc_mat, 50)
```

We compute the embeddings  $U \Sigma$  with the statement `U @ np.diag(s)`.

### 11.5.4 Saving the Vectors

Finally, we store these embeddings in a pandas `DataFrame` with the lines:

```
import pandas as pd

df = pd.DataFrame(
    U @ np.diag(s),
    index=[idx2word[i] for i in range(len(idx2word))])
```

that will index the rows with the words. We save the vectors in a text file with:

```
df.to_csv('cooc50d.txt', sep=' ', header=False)
```

The file consists of rows, where the first item of a row is the word string followed by its embedding: A vector of 50 coordinates separated by spaces.

## 11.6 Embeddings' Similarity

Now that we have counted the cooccurrences and subsequently applied a dimensionality reduction how can we evaluate the results? We saw in Fig. 11.4 that the chapters formed two clusters and that a chapter in French was closer to the other chapters in French than to those in English. We will follow this idea with a few words from our corpus and determine what are the most similar words. If the similar words match our expectations, then we will have a sort of qualitative assessment of the method.

### 11.6.1 Cosine Similarity

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):

$$1 - \cos(\mathbf{u}, \mathbf{v}) = 1 - \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|}.$$

## 11.6.2 Programming

PyTorch has a `nn.CosineSimilarity()` function that takes a row vector  $\mathbf{u}$  and a matrix  $E$  as input and returns a vector of similarities along dimension 1 by default. The next function:

```
def most_sim_vecs(u, E, N=10):
    cos = nn.CosineSimilarity()
    cos_sim = cos(u.unsqueeze(dim=0), E)
    sorted_vectors = sorted(range(len(cos_sim)),
                           key=lambda k: -cos_sim[k])
    return sorted_vectors[1:N + 1]
```

computes the list of the  $N$  vectors most similar to  $\mathbf{u}$  in  $E$  and returns their indices. Note that  $\mathbf{u}$  and  $E$  must be PyTorch tensors and that we convert a NumPy array to a tensor with the function `torch.from_numpy()`.

Taking Homer's *Iliad* and *Odyssey* as corpus, a context of four words, the three most similar words to *he*, *she*, *ulysses*, *penelope*, *achaeans*, and *trojans* are respectively, for the cooccurrence matrix:

```
he ['she', 'it', 'they']
she ['he', 'they', 'ulysses']
ulysses ['achilles', 'hector', 'telemachus']
penelope ['telemachus', 'ulysses', 'juno']
achaeans ['danaans', 'argives', 'trojans']
trojans ['achaeans', 'danaans', 'argives']
```

and after applying the PCA reducing the dimensionality to 50:

```
he ['she', 'it', 'hector']
she ['he', 'minerva', 'juno']
ulysses ['hector', 'menelaus', 'achilles']
penelope ['telemachus', 'neptune', 'ulysses']
achaeans ['danaans', 'argives', 'trojans']
trojans ['sea', 'danaans', 'achaeans']
```

Note that as Facebook's PCA uses a randomized algorithm, these results may vary from run to run.

## 11.7 From Cooccurrences to Mutual Information

We already experimented with semantic associations in Sect. 10.9 and we saw that mutual information could produce relevant pairs. In this section, we will convert our cooccurrence matrix to a matrix of mutual information values. Recall that the definition of mutual information from Fano (1961) is:

$$I(w_i, w_j) = \log_2 \frac{P(w_i, w_j)}{P(w_i)P(w_j)},$$

and we have already the counts in Table 11.2 or in `cooc_mat` from the program in Sect. 11.5.

We will first estimate the probabilities from the counts. We have to consider that each word or word pair appears many more times in the table than in the corpus.

1. To normalize the  $C(w_i, w_j)$  counts, we have to divide them by the sum of all the elements in the matrix. This is just a division by `np.sum(cooc_mat)`;
2. For  $w_i$ , the focus word, its total count is the sum of the elements in row  $i$ .  $P(w_i)$  is then this sum divided by the total count;
3. Finally, for  $w_j$ , the context word, its total count is the sum of the elements in column  $j$ .  $P(w_i)$  is then this sum divided by the total count.

With these counts, we can compute  $\frac{P(w_i, w_j)}{P(w_i)P(w_j)}$  from the cooccurrence matrix.

The next step is to compute the logarithm of the cells. Given the matrix, this is not possible as some pairs  $(w_i, w_j)$  have a count of 0, leading to an infinitely negative value. To solve this problem, Bullinaria and Levy (2007) proposed to set the mutual information of a pair to zero when it is 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.

Finally, we can compute the positive mutual information matrix with this function:

```
def build_pmi_mat(cooc_mat):
    pair_cnt = np.sum(cooc_mat)
    wi_rel_freq = np.sum(cooc_mat, axis=1) / pair_cnt
    wc_rel_freq = np.sum(cooc_mat, axis=0) / pair_cnt

    mi_mat = cooc_mat / pair_cnt # P(w_i, w_j)
    mi_mat /= wc_rel_freq       # P(w_i, w_j)/P(w_j)
    mi_mat = (mi_mat.T / wi_rel_freq).T
                                # P(w_i, w_j)/P(w_j)P(w_i)

    mi_mat = np.log2(mi_mat,
                     out=np.zeros_like(mi_mat),
                     where=(mi_mat != 0))
    mi_mat[mi_mat < 0.0] = 0.0
    return mi_mat
```

where the `where` argument of `np.log2()` tells it to not to apply the function to zero elements and `out` to replace them with 0 instead. We compute the positive mutual information matrix with:

```
pmi_mat = build_pmi_mat(cooc_mat)
```

After computing the positive mutual information on Homer's works, applying a PCA on `pmi_mat` with `fbpca()` reducing the vector dimensionality to 50, and retrieving the most similar words, we obtain this list:

```
he ['him', 'his', 'was']
she ['her', 'herself', 'minerva']
ulysses ['telemachus', 'eumaeus', 'said']
penelope ['telemachus', 'euryklea', 'nurse']
achaeans ['trojans', 'danaans', 'hector']
trojans ['achaeans', 'danaans', 'battle']
```

that seems a bit less relevant than with the previous method.

## 11.8 GloVe

Global Vectors or GloVe (Pennington et al. 2014) is another technique to create embeddings from a corpus. As with the PCA, GloVe's assumption is that words with similar meanings occur in similar contexts. To have a numerical representation of this, the authors extracted cooccurrence counts identical to those in Table 11.2 with contexts of 10 words to the left and 10 words to the right of the focus word. They also weighted the count of each context word by  $1/d$ ,  $d$  being the distance to the focus word as in Sect. 11.5.

### 11.8.1 Model

In Sect. 11.6, we assessed the meaning similarity between the embeddings of two words,  $w_i$  and  $w_j$ , with their cosine. GloVe uses the dot product of the embeddings instead so that it matches how many times  $w_j$  occurs in the vicinity of  $w_i$ . More precisely, GloVe models the dot product of the embeddings a focus word,  $w_i$ , and a word in its context,  $w_j$ , plus two biases, as being equal to the logarithm of their weighted cooccurrence count:

$$E_L(w_i) \cdot E_R(w_j) + b_L(w_i) + b_R(w_j) = \log C(w_i, w_j),$$

where  $E_L(w_i)$  and  $b_L(w_i)$  are the focus word embedding and its bias and  $E_R(w_j)$  and  $b_R(w_j)$ , those of the context word.

## 11.8.2 Loss

The model we described corresponds to an optimization problem that we reformulate with a loss function being the squared error for a pair:

$$(E_L(w_i) \cdot E_R(w_j) + b_L(w_i) + b_R(w_j) - \log C(w_i, w_j))^2$$

and the sum or mean of the squared errors for the whole corpus. We come then to:

$$L = \sum_{i,j=1}^V (E_L(w_i) \cdot E_R(w_j) + b_L(w_i) + b_R(w_j) - \log C(w_i, w_j))^2.$$

We saw in Sect. 11.7 with mutual information that the logarithm could pose a problem as Table 11.2 contained zeros. GloVe solves it by setting the loss to zero when  $C(w_i, w_j) = 0$ . For this, it defines the function:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha}, & \text{if } x < x_{\max}, \\ 1 & \text{otherwise,} \end{cases}$$

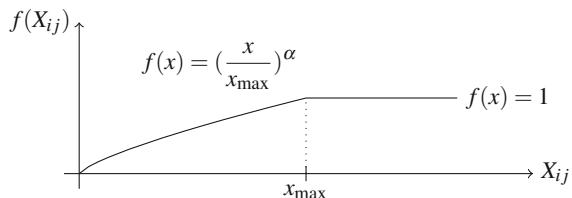
shown in Fig. 11.6 and multiplies it with the squared error. We have thus a zero loss when the counts are zero as  $f(0) = 0$  and by convention  $0 \cdot \log 0 = 0$ . In addition, Pennington et al. (2014) set  $x_{\max}$  to 100 and  $\alpha$  to 3/4.

We have now the final definition of GloVe's loss:

$$L = \sum_{i,j=1}^V f(C(w_i, w_j)) (E_L(w_i) \cdot E_R(w_j) + b_L(w_i) + b_R(w_j) - \log C(w_i, w_j))^2$$

Such a loss enables us to fit the embeddings with a gradient descent and we now describe how to do this with PyTorch.

**Fig. 11.6** GloVe's weighting function with  $\alpha = \frac{3}{4}$



### 11.8.3 Embeddings

The corpus preprocessing step to count the cooccurrences,  $C(w_i, w_j)$ , is exactly the same as in Sect. 11.5. Then, we will store the embedding vectors representing the words in a lookup table as rows of a given dimension. Before we train them, they are just random values.

In PyTorch, we create an embedding table for 9768 words, the vocabulary size, where each word has an embedding dimension of five with the statement:

```
>>> embedding_dim = 5
>>> embedding = nn.Embedding(vocab_size, embedding_dim)
>>> embedding.weight[:5]
tensor([[-1.1903,  0.6513, -0.0738, -0.8198,  1.0269],
       [ 1.7192,  1.9402,  0.8532,  0.0069,  0.1495],
       [ 0.0781, -0.3445, -0.4184,  2.8871, -0.2273],
       [-2.2183, -1.3953,  0.6825,  0.6301,  1.1582],
       [ 0.9579,  0.0194,  0.3023, -0.3885,  0.4006]],

grad_fn=<SliceBackward0>)
```

These parameters are trainable by default.

We then extract embedding vectors with a tensor of indices:

```
>>> word_idx = torch.LongTensor([3, 2, 1])
>>> embedding(word_idx)[:5]
tensor([[-2.2183, -1.3953,  0.6825,  0.6301,  1.1582],
       [ 0.0781, -0.3445, -0.4184,  2.8871, -0.2273],
       [ 1.7192,  1.9402,  0.8532,  0.0069,  0.1495]],

grad_fn=<EmbeddingBackward0>)
```

Once we have trained the embeddings, we will save them and possibly reuse them in other applications. We load trained embeddings with the `from_pretrained()` method:

```
>>> rand_matrix = torch.rand((vocab_size, embedding_dim))
>>> embedding_rand = nn.Embedding.from_pretrained(
    rand_matrix,
    freeze=False)
```

We can make these embeddings trainable or not by setting the `freeze` argument to false or true.

### 11.8.4 The Dataloader

To model the data for the training loop, we will use PyTorch datasets and dataloaders as in Sect. 8.5.3. We will represent the dataset as a `TensorDataset` with two arguments:

1. An input tensor containing the pair indices corresponding to  $(w_i, w_j)$  and
2. An output tensor with the counts representing  $C(w_i, w_j)$ .

We convert the original data structure holding the counts, `C_dict`, see Sect. 11.5, to these tensors with the function `cooc_cnts2Xy()`:

```
def cooc_cnts2Xy(C_dict: dict[dict[int, float]]):
    (C_pairs, C_freqs) = ([], [])
    for word_l, context in C_dict.items():
        for word_r, freq in context.items():
            C_pairs += [[word_l, word_r]]
            C_freqs += [[freq]]
    C_pairs = torch.LongTensor(C_pairs)
    C_freqs = torch.FloatTensor(C_freqs)
    return C_pairs, C_freqs
```

We run it and we create `dataset` and `dataloader` objects with:

```
C_pairs, C_freqs = cooc_cnts2Xy(C_dict)
dataset = TensorDataset(C_pairs, C_freqs)
dataloader = DataLoader(dataset, batch_size=512, shuffle=True)
```

### 11.8.5 Programming the Model

The implementation of the loss is straightforward from its mathematical formula. In the function, we return the loss mean:

```
def loss_fn(Cij_pred, Cij, alpha=0.75, x_max=100):
    weight_function = torch.pow(Cij/x_max, alpha)
    weight_function[weight_function > 1] = 1.0
    loss = weight_function * (Cij_pred - torch.log(Cij))**2
    return loss.mean()
```

We derive a `nn.Module` class to compute the embedding products. The initialization creates two embedding tables whose sizes are the number of words by the embedding dimension. It also creates two column embeddings containing the biases. The `forward()` function computes the dot product of  $w_i$  and  $w_j$  for a batch of pairs and adds the biases.

```
class GloVe(nn.Module):
    def __init__(self, vocab_size, d, glove_init=True):
        super().__init__()
        self.E_l = nn.Embedding(vocab_size, d)
        self.E_r = nn.Embedding(vocab_size, d)
        self.b_l = nn.Embedding(vocab_size, 1)
        self.b_r = nn.Embedding(vocab_size, 1)

    def forward(self, Pij):
        focus = Pij[:, 0]
        cword = Pij[:, 1]
        batched_dot = (
            (self.E_l(focus) *
             self.E_r(cword)).sum(dim=-1, keepdim=True)
            + self.b_l(focus) + self.b_r(cword))
        return batched_dot
```

Finally, we create the model and the optimizer with

```
glove = GloVe(vocab_size, d)
optimizer = torch.optim.NAdam(glove.parameters())
```

and we fit the parameters with a training loop identical to that in Sect. 8.5.3.

### 11.8.6 Computing the Embeddings

Once fitted, our model contains two embedding tables: one for the focus words and the other for the context words. Pennington et al. (2014) proposed to add them to get the GloVe embeddings:

```
E = glove.E_l.weight + glove.E_r.weight
```

We can store these vectors in a pandas `DataFrame` as in Sect. 11.5.4. We must though detach the tensors from their computational graph and convert them to NumPy with the method: `E.detach().numpy()`. We then save them in a file as we did in this section.

### 11.8.7 Semantic Similarity

Using the cosine similarity of Sect. 11.6, the same corpus, and the same list of words, we obtain:

```
he ['but', 'was', 'him']
she ['he', 'her', 'then']
ulysses ['telemachus', 'achilles', 'said']
penelope ['telemachus', 'juno', 'answered']
trojans ['achaeans', 'danaans', 'the']
achaeans ['trojans', 'suitors', 'danaans']
```

for vectors of 50 coordinates, trained on 50 epochs with a context of four words to the left and to the right.

## 11.9 Word Embeddings from Neural Networks

### 11.9.1 word2vec

After the PCA and GloVe, word2vec (Mikolov et al. 2013a) is another technique to obtain embeddings. It uses neural networks and comes in two forms: CBOW and skipgrams. The goal of CBOW is to guess a missing word given its surrounding context as in the example below:

Sing, O goddess, the anger of Achilles son of Peleus,

where a reader, given the incomplete phrase:

`Sing, 0 ----, the anger of Achilles`

would have to fill in the blank with the word `goddess`. This set up is then identical to fill-in-the-blank questionnaires or *cloze tests* (Taylor 1953).

In machine learning, this task corresponds to a prediction. The  $\mathbf{x}$  input, called the context, is a word sequence deprived from one word. This missing word or target is the  $\mathbf{y}$  answer. We can easily create a dataset from any text. Using our example above and sequences of five words, we generate the  $X$  contexts by removing the word in the middle and we build the  $\mathbf{y}$  vector with the words to predict:

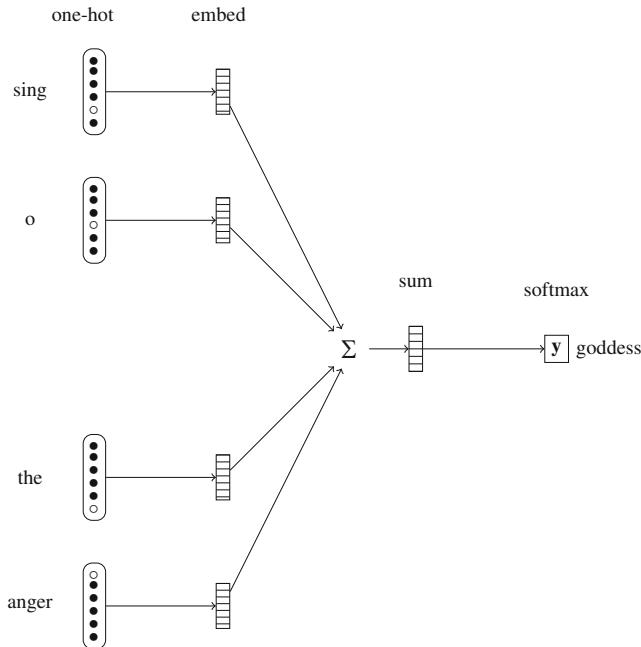
$$X = \begin{bmatrix} \text{sing} & \text{o} & \text{the} & \text{anger} \\ \text{o} & \text{goddess} & \text{anger} & \text{of} \\ \text{goddess} & \text{the} & \text{of} & \text{achilles} \\ \text{the} & \text{anger} & \text{achilles} & \text{son} \\ \text{anger} & \text{of} & \text{son} & \text{of} \\ \text{of} & \text{achilles} & \text{of} & \text{peleus} \end{bmatrix}; \mathbf{y} = \begin{bmatrix} \text{goddess} \\ \text{the} \\ \text{anger} \\ \text{of} \\ \text{achilles} \\ \text{son} \end{bmatrix}$$

skipgrams is the second and symmetrical part of word2vec. Instead of predicting a word from its context, it predicts the context from a word. In the two next sections, we will design architectures to generate word2vec embeddings with the CBOW setup and then with skipgrams.

### 11.9.2 CBOW Architecture

While the previous techniques used cooccurrences, word2vec starts from an input of one-hot encoded words, in the form of indices, and carries out the dimensionality reduction inside a neural network. To compute the embeddings, we first train a model to predict the words from their contexts and then extract the model's input weights that will correspond to the CBOW embeddings.

Given a training corpus, the first step extracts the vocabulary of its words. The network input consists of the context words encoded by their indices in the vocabulary followed by an **embedding layer**. This layer is just a lookup mechanism, as we saw in Sect. 11.8.3, that replaces the word indices with their dense representations: Vectors of the embedding dimension. Practically, a dense representation is a trainable vector of 10–300 dimensions. The next layer computes the sum or the mean of the embedding vectors resulting into one single vector. Finally, a linear layer and a softmax function predict the target word. This last layer has as many weights as we have dimensions in the embeddings; see Fig. 11.7.



**Fig. 11.7** The CBOW architecture. After Mikolov et al. (2013a)

We initialize randomly the dense vector representations of the words. The vector parameters are then learned by the fitting procedure. With this technique, the word embeddings correspond to the weights of the first layer of the network.

### 11.9.3 Programming CBOW

#### Preprocessing the Text

In Fig. 11.7, the input consists of five words or batches of five words. We have first to format our data so that it fits this input. We use the same statements as in Sect. 11.5.1 to tokenize the words and create the indices. We apply them on a corpus made of a concatenation of the *Iliad* and the *Odyssey*.

Then given a context size `c_size` of 5 and left and right contexts `w_size` of 2, we create the list of contexts in `X` and of words to predict in `y` with this function:

```
def create_Xy(words):
    (X, y) = ([], [])
    c_size = 2 * w_size + 1
    for i in range(len(words) - c_size + 1):
        X.append(words[i: i + w_size] +
                  words[i + w_size + 1: i + c_size])
        y.append(words[i + c_size])
```

```

    y.append(words[i + w_size])
return X, y

```

We apply this function to the list of words and we obtain the same values as in the description of CBOW for  $X$ :

```

>>> X, y = create_Xy(words)
>>> X[2:5]
[[‘sing’, ‘o’, ‘the’, ‘anger’],
 [‘o’, ‘goddess’, ‘anger’, ‘of’],
 [‘goddess’, ‘the’, ‘of’, ‘achilles’]]

```

and  $y$ :

```

>>> y[2:5]
[‘goddess’, ‘the’, ‘anger’]

```

To create the equivalent tensors, we need numerical values. We call `create_Xy()` with the list of indices, `words_idx`, as input instead of the words:

```
X, y = create_Xy(words_idx)
```

and we convert  $X$  and  $y$  to PyTorch tensors, the input and target values:

```

X = torch.LongTensor(X)
y = torch.LongTensor(y)

```

The words are now replaced with indices in  $X$ :

```

>>> X[2:5]
tensor([[7663, 5691, 8548, 358],
        [5691, 3697, 358, 5735],
        [3697, 8548, 5735, 67]])

```

and  $y$ :

```

>>> y[2:5]
tensor([3697, 8548, 358])

```

## Embeddings

The first layer in Fig. 11.7 consists of an embedding layer. This is just a lookup table, as we saw in Sect. 11.8.3, where we store the embedding vectors as rows of a given dimension. Before we train them, they are just random values.

We extract the embeddings of the five first contexts with the statement:

```

>>> embedding(X[2:5])
>>> embedding(X[2:5]).size()
torch.Size([3, 4, 5])

```

where the first axis corresponds to the batch size, here 3, from  $x[2]$  to  $x[4]$ , the second one to the size of the input, here four words making the context, and the third one, the dimension of the embedding vectors: 5.

In Fig. 11.7, we sum or compute the mean of the embeddings making up the context. We simply add `sum()` or `mean()` to the tensor with the dimension where we apply the operation, here the four context words:

```
>>> embedding(X[2:5]).mean(dim=1)
tensor([[ 0.0108, -0.6135,  0.1026,  0.0504,  0.3282],
       [ 0.7513,  0.1777, -0.0256, -0.1262,  0.7771],
       [ 0.2507,  0.1822, -0.1035,  0.1529,  0.4330]], 
grad_fn=<MeanBackward1>)
```

## Embedding Bags

As the sum or mean of embeddings is a frequent operation, PyTorch has a dedicated embedding class that computes it automatically, `EmbeddingBag`, by default the mean.

```
>>> embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim)
```

This statement creates a new embedding matrix. To have the same content as our first embeddings, we load our first embedding matrix:

```
embedding_bag = nn.EmbeddingBag.from_pretrained(
    embedding.state_dict()['weight'])
```

The result is then the same as with `mean()` in the previous section:

```
>>> embedding_bag(X[2:5])
tensor([[ 0.0108, -0.6135,  0.1026,  0.0504,  0.3282],
       [ 0.7513,  0.1777, -0.0256, -0.1262,  0.7771],
       [ 0.2507,  0.1822, -0.1035,  0.1529,  0.4330]])
```

## The Network

It is now easy to program the architecture in Fig. 11.7 with PyTorch. We just declare the sequence with the tensor dimensions:

```
embedding_dim = 50
model = nn.Sequential(
    nn.EmbeddingBag(vocab_size, embedding_dim),
    nn.Linear(embedding_dim, vocab_size))
```

We select a loss and an optimizer:

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.NAdam(model.parameters(), lr=0.025)
```

and finally, we create a dataset and a dataloader:

```
dataset = TensorDataset(X, y)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

The training loop is then identical to that in Sect. 8.8.

## CBOW Embeddings

We save the whole model with:

```
torch.save(model, 'cbow.model')
```

In fact, only the embeddings are really interesting. They correspond to the first layer of the model. We extract it with:

```
model[0].weight
```

We can store these embeddings in a pandas `DataFrame` as in Sect. 11.5.4 and save them in a file.

## Semantic Similarity

Once our model is trained, we can compute the cosine similarity of a few words using the function in Sect. 11.6. Using a batch size of 1024, the `mean()` function to aggregate the vectors, and five epochs, the five most similar words for: *he*, *she*, *ulysses*, *penelope*, *achaeans*, and *trojans* are respectively:

```
he ['she', 'they', 'i']
she ['he', 'vulcan', 'they']
ulysses ['telemachus', 'antinous', 'juno']
penelope ['helen', 'telemachus', 'juno']
achaeans ['danaans', 'trojans', 'argives']
trojans ['danaans', 'achaeans', 'argives']
```

which all correspond to semantically related terms. These lists may vary depending on the network initialization.

### 11.9.4 Skipgrams

#### The Model

In the skipgram task, we try to predict the context of a given word, i.e. the words surrounding it. For instance, taking the word *goddess* in:

*Sing, O goddess, the anger of Achilles son of Peleus,*

and a window of two words to the left and to the right of it, we would predict: *Sing*, *O*, *the*, and *anger*.

For this, the skipgram model tries to maximize the average log-probability of the context. For a corpus consisting of a sequence of  $T$  words,  $w_1, w_2, \dots, w_T$ , this yields:

$$\frac{1}{T} \sum_{t=c+1}^{T-c} \sum_{\substack{j=-c, \\ j \neq 0}}^c \log P(w_{t+j}|w_t),$$

where  $c$  is the context size. Using our example and the word *goddess*, this would correspond to the sum of logarithms of:  $P(\text{sing}|\text{goddess})$ ,  $P(\text{o}|\text{goddess})$ ,  $P(\text{the}|\text{goddess})$ , and  $P(\text{anger}|\text{goddess})$ .

Given an input word,  $w_i$ , for instance *goddess*, we compute the probability  $P(w_o|w_i)$  of word  $w_o$  to be in its context using the dot product of their respective embeddings,  $\mathbf{v}_{w_i}$  for the center word and  $\mathbf{v}'_{w_o}$  for the context. This dot product,  $\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_o}$ , reflects the proximity of the two embeddings. We then use the softmax function to normalize the values and get a probability sum of one:

$$P(w_o|w_i) = \frac{\exp(\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_o})}{\sum_{w \in V} \exp(\mathbf{v}_{w_i} \cdot \mathbf{v}'_w)},$$

where  $V$  is the vocabulary.

### Skipgrams Negative Sampling

The skipgram initial setting results in a model with a huge number of parameters. To reduce it, Mikolov et al. (2013b) proposed a binary classification, where they extracted pairs consisting of a word and a context word, forming the positive class, as well as pairs, where the second word is outside the context, the negative class. As classes, we have then  $y = 1$ , when the words are cooccurring, and  $y = 0$ , when not.

We represent the words with their embeddings and we compute the dot product of vector pairs consisting of the input word and a word in the context. For the negative class, we use a pair consisting of the input word and a word drawn randomly from the corpus.

Using our *goddess* example again and a window of two words to the left and to the right of it, we have the positive pairs:

$$X = \begin{bmatrix} \text{goddess sing} \\ \text{goddess o} \\ \text{goddess the} \\ \text{goddess anger} \end{bmatrix}; \mathbf{y} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

For each positive pair, Mikolov et al. (2013b) proposed to draw randomly  $k$  negative pairs, for instance *labours*, *to*, *end*, *before*, *and*, etc.

$$X = \begin{bmatrix} \text{goddess labours} \\ \text{goddess to} \\ \text{goddess end} \\ \text{goddess before} \\ \text{goddess ...} \end{bmatrix}; \mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \dots \end{bmatrix}$$

where  $k$  is a parameter between 2 and 25.

The dot product ranges from  $-\infty$  to  $\infty$ . To have a logarithmic loss compatible with a neural net architecture, we use the logistic curve as activation function. See the loss below.

## The Network

From these definitions, we can program a network. Contrary to the CBOW model, the architecture is not a sequence and we have to define a new class as in Sect. 8.5.3. We have a pair consisting of an input word, `i_word`, and a context word that corresponds to the output `o_word` in the network. Both words go through their respective embedding layers that we define in the `__init__` method. In the forward method, we split the  $X$  input in two columns and we compute the dot product of the pairwise vectors:

```
class Skipgram(nn.Module):
    def __init__(self):
        super().__init__()
        self.i_embedding = nn.Embedding(vocab_size,
                                        embedding_dim)
        self.o_embedding = nn.Embedding(vocab_size,
                                        embedding_dim)

    def forward(self, X):
        i_embs = self.i_embedding(X[:, 0])
        o_embs = self.o_embedding(X[:, 1])
        x = (i_embs * o_embs).sum(dim=-1, keepdim=True)
        return x
```

## The Loss

To train the model, we need a loss. Mikolov et al. (2013b) defined it from the network output, the dot product of the input and output embeddings, as:

$$-\log \sigma(\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_o}) - \sum_{i=1}^k \log \sigma(-\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_n}),$$

where  $\sigma$  is the logistic function,  $w_i$  is the center word,  $w_o$ , a word in the context, and  $w_n$ , a word outside it. This loss reflects the property that words in the context should be similar to the center word, while the other words should be dissimilar.

There is no such a built-in loss in PyTorch, so we need to define it. In Sect. 11.8.5, we wrote a function to compute a custom loss. Here, we will see another way to do it with a class that applies to a batch with the same signature: `loss_fn(y_pred, y)` and returns the mean or the sum for the batch.

The `y_pred` vector contains the predictions for the positive and negative classes. We compute two intermediate loss vectors: For class 1,  $\log \sigma(\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_o})$ ; for class

$0, \log \sigma(-\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_n})$ . We then extract the values relevant to the positive class by a pointwise multiplication with  $\mathbf{y}$ :

$$\mathbf{y} \odot \log \sigma(\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_o}),$$

and for the negative class with  $\mathbf{1} - \mathbf{y}$ :

$$(\mathbf{1} - \mathbf{y}) \odot \log \sigma(-\mathbf{v}_{w_i} \cdot \mathbf{v}'_{w_n}).$$

We add these two vectors and we compute the mean to have the loss. This corresponds to the class:

```
class NegSamplingLoss(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, y_pred, y):
        p = y * torch.log(torch.sigmoid(y_pred))
        n = (1.0 - y) * torch.log(torch.sigmoid(-y_pred))
        return -(p + n).mean(dim=0)
```

We create the loss, the model, and the optimizer with the statements:

```
model = Skipgram()
loss_fn = NegSamplingLoss()
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001)
```

## Dataset Preparation

As with CBOW, we tokenize the corpus and we extract the  $(w_i, w_o)$  pairs by traversing the list. Then we have to build the negative pairs:  $(w_i, w_n)$ . We then draw randomly from their distribution. We can do this with the function:

```
random.choices(word_idx, weights=frequencies, k=N)
```

that will return  $k$  indices according to the `weights` distribution.

To derive the distribution, we have to count the words and divide them by the size of the corpus resulting in a probability distribution  $U$ . Nonetheless, Mikolov et al. (2013b) noticed that the application of a power to  $U$  yielded better results:

$$U(w) = \frac{C(w)^{\text{power}}}{\sum_i C(w_i)^{\text{power}}}$$

We apply a power transform to the initial counts or distribution with the function:

```
def power_transform(dist, power):
    dist_pow = {k: math.pow(v, power)
               for k, v in dist.items()}
    total = sum(dist_pow.values())
    dist_pow = {k: v/total}
```

```

        for k, v in dist_pow.items()}
    return dist_pow

```

and we return power transformed probabilities. We use  $\frac{3}{4}$  as power.

With these functions, we can build the  $X$  and  $y$  matrices of indices, create datasets and dataloaders, and run the training loop. Mikolov et al. (2013b) also proposed to downsample the frequent words. To carry this out, they discarded each word  $w$  in their corpora with the probability:

$$P(w) = 1 - t \sqrt{\frac{\sum_i C(w_i)}{C(w)}}$$

with  $t \approx 0.003$ .

## Semantic Similarity

When trained on Homer's *Iliad* and *Odyssey* on five epochs, the three most similar words to those in our list are:

```

he ['she', 'they', 'it']
she ['he', 'her', 'him']
ulysses ['achilles', 'telemachus', 'hector']
penelope ['speak', 'nestor', 'telemachus']
achaeans ['trojans', 'argives', 'two']
trojans ['achaeans', 'argives', 'cyprian']

```

## 11.10 Application of Embeddings to Language Detection

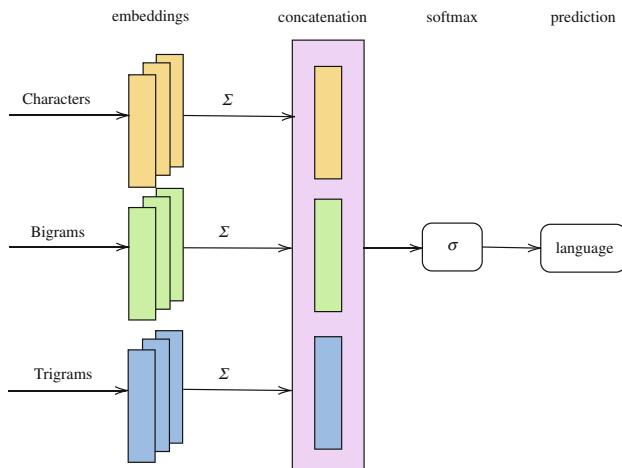
Language detection is an example of categorization, where we predict the language of a text or a sentence, as we have seen in Sects. 8.5 and 8.6. So far, we have used letter counts as input to a logistic regression. As an application of what we have learned in the chapter, we will train a model, where the input consists of embeddings of character  $n$ -grams instead. This model is similar to Google's publicly available compact language detector (CLD3).<sup>1</sup>

This model represents a more realistic application case than those we have seen earlier in the chapter. Its development will require a longer program and we will split it into smaller tasks, where we will:

1. Select and preprocess a dataset;
2. Balance it;

---

<sup>1</sup> GitHub repository: <https://github.com/google/cld3>.



**Fig. 11.8** Google’s CLD3 architecture

3. Encode the characters, character bigrams and trigrams of a text using embedding tables and compute their mean. This corresponds to the left part of Fig. 11.8;
4. Create a logistic regression model that takes a concatenation of the embedding means and outputs the language code; see the middle and right part of Fig. 11.8;
5. Train this model, i.e the embedding tables and the logistic layer, and report the results.

### 11.10.1 Corpus

This first step before we train a model is to find a dataset. We will use that of Tatoeba, a collaborative site, where users add or translate short texts and annotate them with language tags. In the end of year 2023, the Tatoeba corpus reached nearly 12 million texts in more than 400 languages. It is available in one file called `sentences.csv`.<sup>2</sup> Many applications use Tatoeba to train machine-learning models including translation and language detection.

The dataset is structured this way: There is one text per line, where each line consists of a unique identifier, the text, and the language code:

```
sentence identifier [tab] language code [tab] text [cr]
```

The fields are separated by tabulations and ended by a carriage return. The language codes follow an ISO standard: `eng` for English, `fra` for French, `cnn` for Chinese, `deu` for German, etc. Here are a few lines from the dataset:

---

<sup>2</sup> Tatoeba download site: <https://tatoeba.org/eng/downloads>.

```
('1276', 'eng', "Let's try something.")
('1277', 'eng', 'I have to go to sleep.')

('3091', 'fra', 'Essayons quelque chose\u202f!')
('3093', 'fra', "Qu'est-ce que tu fais\u202f?")

('6057', 'deu', 'Möchten Sie eine Tasse Kaffee?')
('6058', 'deu', 'Ich gehe in einem Park spazieren.')
```

### 11.10.2 Preprocessing and Balancing the Dataset

The Tatoeba dataset is very large and, at the same time, some languages only have a handful of texts. We will extract a working dataset from it with languages that have at least 20,000 sentences. We will then downsample this data so that we have an equal number of sentences for each language. We will finally split it into training, validation, and test sets.

We first read the sentences with a generator:

```
def file_reader(file):
    with open(file, encoding='utf8', errors='ignore') as f:
        for line in f:
            row = line.strip()
            yield tuple(row.split('\t'))

line_generator = file_reader('sentences.csv')
```

and we count the texts per language to enable us to keep the languages with enough training data:

```
>>> lang_freqs = Counter(map(lambda x: x[1], line_generator))
>>> lang_freqs.most_common(3)
[('eng', 1854349), ('rus', 1027631), ('ita', 867469)]
```

We select the languages with more than 20,000 samples with the statements:

```
SENT_PER_LANG = 20000

selected_langs = {lang: freq for lang, freq
                 in lang_freqs.items()
                 if freq > SENT_PER_LANG}
```

Classifiers tend to bias their choice toward the most populated classes. To counter this, we will balance the corpus and we will keep about 20,000 texts for each language. This corresponds to a fraction of the Tatoeba corpus that depends on the language. English has 1,854,349 samples and we will select  $\frac{20,000}{1,854,349} = 0.011\%$  of them; Russian has 1,027,631 and we will select  $\frac{20,000}{1,027,631} = 0.019\%$ , etc.

To choose the samples, we will use a uniform generator of random numbers between 0 and 1. We will scan the sentences of our dataset and, for each of them, draw a random number. If we want to select 10% of the sentences of a given

language, we will return the sentences for which the random number is less than 0.1, for 20%, when it is less than 0.2, etc. In our case, we compute the percentages so that for each language we have 20,000 texts.

```
lang_percentage = dict()
for lang, cnt in selected_langs.items():
    lang_percentage[lang] = SENT_PER_LANG/cnt
```

We can now extract the sentences of our working corpus and shuffle it:

```
line_generator = file_reader('sentences.csv')

working_corpus = []
for lang_tuple in line_generator:
    lang = lang_tuple[1]
    if (lang in lang_percentage and
        random.random() < lang_percentage[lang]):
        working_corpus += [lang_tuple]

random.shuffle(working_corpus)
```

We split it into training, validation, and test sets with the ratios: 80%, 10%, and 10%:

```
TRAIN_PERCENT = 0.8
VAL_PERCENT = 0.1
TEST_PERCENT = 0.1

split_sizes = list(map(lambda x: int(len(working_corpus) * x),
                      (TRAIN_PERCENT, VAL_PERCENT, TEST_PERCENT)))
```

This results in datasets of respectively 656,871, 82,108, and 82,108 samples. We compute the corresponding offsets in the working corpus with:

```
offsets = [sum(split_sizes[:i])
           for i in range(len(split_sizes))]
```

and we extract our sets from it:

```
training_set = working_corpus[:offsets[1]]
val_set = working_corpus[offsets[1]:offsets[2]]
test_set = working_corpus[offsets[2]:]
```

We also store them as pandas DataFrames:

```
dp_train = pd.DataFrame(working_corpus[:offsets[1]])
dp_val = pd.DataFrame(working_corpus[offsets[1]:offsets[2]])
dp_test = pd.DataFrame(working_corpus[offsets[2]:])
```

that we can save as in Sect. 4.3.

```
dp_train.to_csv('tatoeba.train', sep='\t',
                header=False, index=False)
```

### 11.10.3 Character N-Grams

Google's CLD3 uses characters as input, as well as character bigrams and trigrams, each with its own embedding table. We extract such  $n$ -grams from a string with a given  $n$  value with this function:

```
def ngrams(sentence: str,
           n: int = 1) -> list[str]:
    ngram_1 = []
    for i in range(len(sentence) - n + 1):
        ngram_1 += [sentence[i:i+n]]
    return ngram_1
```

and we collect all the  $n$ -grams from  $n = 1$  up to 3 with this function:

```
def all_ngrams(sentence: str,
               max_ngram: int = 3,
               lc=True) -> list[list[str]]:
    if lc:
        sentence = sentence.lower()
    all_ngram_list = []
    for i in range(1, max_ngram + 1):
        all_ngram_list += [ngrams(sentence, n=i)]
    return all_ngram_list
```

Applied to the training set, this yields more than 5000 unique characters, 100,000 unique bigrams, and 340,000 trigrams as there are many nonLatin alphabets. In Sect. 4.1.2, we saw that Unicode has a coding capacity of 24 bits resulting in a ceiling of more than 1,000,000 different characters. This makes it impossible to use one-hot encoding in such an application.

### 11.10.4 Encoding the N-Grams

We will use a hash function to reduce these numbers with a technique sometimes called the *hashing trick*. Hash functions map a data object to a fixed length number such as Python's built-in hash function:

```
>>> hash('a')
7642859311653074128
>>> hash('ac')
9165824848673741814
>>> hash('abc')
29577857799253695
```

We can reduce the hash code range of the  $n$ -grams by dividing them by a fixed integer, the *modulus*, for instance 100, and taking the remainder or *modulo*, resulting then in at most 100 different codes.

```
>>> MOD_CHARS = 100
>>> hash('a') % MOD_CHARS
```

This hashing technique has many advantages here as we can skip the  $n$ -gram indexing: We just need to apply the hash function to any  $n$ -gram. In addition, there is no unknown value as it will always return an integer lower than the modulus. Nonetheless, hashing will also create encoding conflicts as the coding capacity is reduced from Unicode's 1 million characters to the modulus. In our example, as we have about 5000 different characters, this means that on average, with a modulus of 100, 50 characters will have the same hash value. We will see that this will not harm the classification results.

As modulus for the characters, bigrams, and trigrams, we will use the prime numbers `MODS = [2053, 4099, 4099]` as they usually have better distributional properties.

A problem with Python's hashing function is that it does not always return the same values when executed on different machines. We replace it with the MD5 standard and this function instead (Klang 2023):

```
import hashlib
def reproducible_hash(string: str) -> int:
    h = hashlib.md5(string.encode('utf-8'),
                    usedforsecurity=False)
    return int.from_bytes(h.digest()[0:8], 'big', signed=True)
```

where `hashlib.md5()` creates a hash encoder and `digest()` computes the code in the form of bytes. We then return an integer from it.

We can now convert a  $n$ -gram to a numerical code. We write a function to apply this to a list of strings, where the modulus is an argument:

```
def hash_str_list(text: list[str],
                  char_mod: int) -> list[int]:
    values = map(lambda x: x % char_mod,
                map(reproducible_hash, text))
    return list(values)
```

### 11.10.5 Building X and y

We will now create our input and output matrices. We will only write the code for the training set as this is the same for the two other sets. First, we create indices for the languages:

```
idx2lang = dict(enumerate(sorted(selected_langs)))
lang2idx = {v: k for k, v in idx2lang.items()}
```

Then we create a tensor of the y output:

```
y_train = torch.LongTensor(
    [lang2idx[tuple_lang[1]]
     for tuple_lang in training_set])
```

Now we can look at the  $X$  matrix. We first create a function that, given a text, creates the  $n$ -grams and convert them into hash codes. We return three tensors of indices:

```
def hash_all_ngrams(text: str,
                     max_ngram=N_MAX,
                     mod_list=MODS):
    all_ngrams_1 = all_ngrams(text, max_ngram)
    x = []
    for ngram_1, mod in zip(all_ngrams_1, mod_list):
        x += [torch.LongTensor(hash_str_list(ngram_1, mod))]
    return x
```

To build the  $X$  matrix, we just call this hash function on all the texts of the training set. We carry this out with the `build_X()` function:

```
def build_X(dataset):
    X = []
    for lang_tuple in tqdm(dataset):
        x = hash_all_ngrams(lang_tuple[2])
        X += [x]
    return X
```

and we have

```
X_train = build_X(training_set)
```

`X_train` is a list of triples, where each triple represents one text and consists of three tensors of indices for the characters, bigrams, and trigrams.

### 11.10.6 Bags of Embeddings

The language detector associates each character of a text to an embedding and computes the means of the embeddings; see Fig. 11.8. It also does this for the bigrams and trigrams and concatenates the three tensors. Finally, it passes the result to the logistic regression layer. We saw in Sect. 11.9.3 how to create an embedding bag that computes the mean. However, we have here a different numbers of characters or  $n$ -grams across the texts. This would then make it impossible to process batches of inputs as they must be rectangular. Fortunately, it is possible with PyTorch to create embedding bags of variable length. We then pass a 1D list of indices and the offsets marking the position of each bag as second argument.

Let us exemplify this with a small embedding table of three symbols and embeddings of dimension 5:

```
embs = nn.EmbeddingBag(3, 5)
```

We look up two lists of embeddings and compute their means with:

```
>>> embs(torch.LongTensor([[0, 1, 2], [2, 1, 0]]))
tensor([[-0.2384,  0.4318,  0.1966, -0.3318, -0.5874],
       [-0.2384,  0.4318,  0.1966, -0.3318, -0.5874]],
       grad_fn=<EmbeddingBagBackward0>)
```

which is equivalent to a flat list of indices and a list of offsets:

```
>>> embs(torch.LongTensor([0, 1, 2, 2, 1, 0]),
        torch.LongTensor([0, 3]))
tensor([[-0.2384,  0.4318,  0.1966, -0.3318, -0.5874],
       [-0.2384,  0.4318,  0.1966, -0.3318, -0.5874]],
      grad_fn=<EmbeddingBagBackward0>)
```

With such offsets, we can create a batch of samples that do not need a rectangular table of indices.

We can now create a function that generates a bag for a batch of hash codes. Given a list of tensors of indices as input, we concatenate the tensors, we compute their lengths, and then their offsets with this function:

```

def bag_generator(X_idx_l: list):
    X_idx = torch.cat(X_idx_l, dim=-1)
    bag_lengths = [X_idx_l[i].size(dim=0)
                   for i in range(len(X_idx_l))]
    X_offsets = [sum(bag_lengths[:i])
                 for i in range(len(bag_lengths))]
    return X_idx, torch.LongTensor(X_offsets)

```

We return the flat tensor and the offsets.

To process the whole  $X$  table, we need a somehow convoluted statement:

```
bags_idx, bags_offsets = zip(*map(bag_generator, zip(*X_train[0:5])))
```

A batch in the  $X$  table, for instance  $x[:, 5]$  is a stack of triples: the characters, bigrams, and trigrams. We have to generate a separate bag each of them. The inside `zip` extract the three columns, corresponding respectively to the characters, the bigrams, and the trigrams and apply them the bag generator. We have then three bags made of a pair of the indices and the offsets. The second `zip` creates two lists: one for the indices and the other for the offsets.

### 11.10.7 Model

The model follows the language detector architecture, where we first look up the embeddings for the characters, bigrams, and trigrams, compute their respective means with an embedding bag, concatenate these means, and then pass the vector to a linear layer:

```

self.fc = nn.Linear(3 * EMBEDDING_DIM, n_lang)

def forward(self, bags, offsets):
    (bags1, bags2, bags3) = bags
    (offsets1, offsets2, offsets3) = offsets
    x1 = self.emb_chars(bags1, offsets1)
    x2 = self.emb_bigr(bags2, offsets2)
    x3 = self.emb_trig(bags3, offsets3)
    x = torch.hstack((x1, x2, x3))
    x = self.fc(x)
    return x

```

We create a language detectors object with:

```
model = LangDetector(MODS, EMBEDDING_DIM, len(lang2idx))
```

### **11.10.8 Training Loop**

We can now train the model. We select a loss and an optimizer:

```
loss_fn = nn.CrossEntropyLoss()      # cross entropy loss
optimizer = torch.optim.NAdam(model.parameters(), lr=0.01)
```

The training loop is identical to those we have already written. The only difficulty being the call to the bag generator and we detail it here. We skip the piece of code to measure the performance on the validation set.

```

loss_train_history = []
acc_train_history = []

for epoch in tqdm(range(5)):
    loss_train = 0
    acc_train = 0
    cnt = 0
    model.train()
    n_indices = list(range(0, len(X_train), BATCH_SIZE))
    random.shuffle(n_indices)
    for i in n_indices:
        X_train_bags_idx, X_train_bags_offsets = zip(
            *map(bag_generator,
                  zip(*X_train[i:i + BATCH_SIZE])))

        y_batch = y_train[i:i + BATCH_SIZE]
        y_pred_batch = model(X_train_bags_idx,
                             X_train_bags_offsets)
        loss_batch = loss_fn(y_pred_batch, y_batch)
        optimizer.zero_grad()
        loss_batch.backward()
        optimizer.step()
        loss_train += loss_batch.item()
        acc_train += torch.sum(
            y_batch == torch.argmax(y_pred_batch,
                                   dim=-1))/BATCH_SIZE
    loss_train_history.append(loss_train)
    acc_train_history.append(acc_train)

```

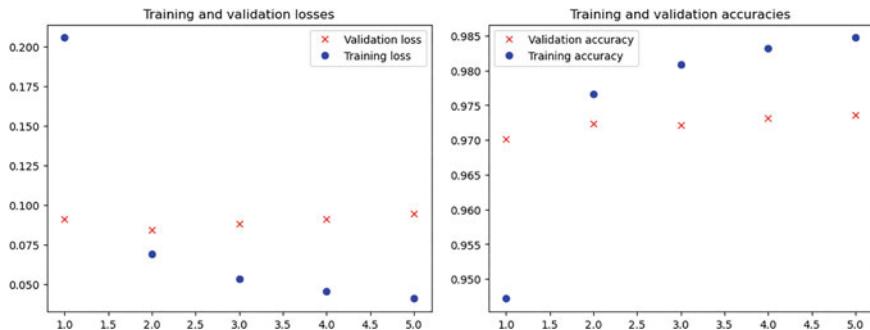


Fig. 11.9 The training loss and accuracy over the epochs

```

acc_train /= len(n_indices)
acc_train_history += [acc_train.item()]
loss_train /= len(n_indices)
loss_train_history += [loss_train]

```

### 11.10.9 Evaluation

The curves in Fig. 11.9 show the training and validation losses and accuracies over the epochs. As we can see, training and validation figures start diverging after two epochs, hinting at an overfit as soon as epoch 3. Running the fitting loop again and stopping after two epochs, we obtain a macro average of 0.9723 for the validation set and of 0.9735 for the test set.

## 11.11 Further Reading

The cooccurrences matrices we saw in Sect. 11.2 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,b), and especially Benzécri (1981b), which 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.

In this chapter, we saw word embeddings based on mutual information, word2vec (Mikolov et al. 2013a), and Glove (Pennington et al. 2014). fastText (Bojanowski et al. 2017) is another popular method. All these systems use

iterative training procedures and can handle larger corpora than singular value decomposition.

As the power of computers and graphical processing units (GPU) increased dramatically, word embeddings found new applications. Most notably, replacing input words with pre-trained embeddings is an efficient way to mitigate sparse data. A big advantage is that we can train these embeddings on large raw corpora, where they capture semantic regularities. This explains why they are now ubiquitous in classification and other NLP models.

Finally, we created pedagogical programs to understand their inner workings. The original programs of word2vec, Glove, and fastText are open source and optimized. They are probably better choices for production applications. In addition, the authors trained them on large corpora and published their vectors. This makes it possible to reuse them as pretrained parameters in neural network models. We will see an application of this with GloVe vectors in Chap. 14, *Part-of-Speech and Sequence Annotation*.

# Chapter 12

## Words, Parts of Speech, and Morphology



Partes orationis quot sunt? Octo. Quae? Nomen, pronomen, verbum, adverbium, participium, coniunctio, praepositio, interiectio.  
Aelius Donatus, *Ars grammatica. Ars minor*, Fourth century.

### 12.1 Words

#### 12.1.1 *Parts of Speech*

In the previous chapters, when we processed the words, we did not make any distinction on their possible intrinsic character. As words have different grammatical properties, classical linguists designed classes to gather those sharing common properties. They called these classes **parts of speech** (POS). The concept of part of speech dates back to the classical antiquity philosophy and teaching. Plato made a distinction between the verb and the noun. After him, the word classification further evolved, and parts of speech grew in number until Dionysius Thrax fixed and formulated them in a form that we still use today. Aelius Donatus popularized the list of the eight parts of speech: noun, pronoun, verb, adverb, participle, conjunction, preposition, and interjection, in his work *Ars grammatica*, a reference reading in the Middle Ages.

The word parsing comes from the Latin phrase *partes orationis* ‘parts of speech’. It corresponds to the identification of the words’ parts of speech in a sentence. In natural language processing, POS tagging is the automatic annotation of words with grammatical categories, also called POS tags. Parts of speech are also sometimes called lexical categories.

Most European languages have inherited the Greek and Latin part-of-speech classification with a few adaptations. The word categories as they are taught today roughly coincide in English, French, and German in spite of some inconsistencies.

**Table 12.1** Closed class categories

Part of speech	English	French	German
Determiners	<i>the, several, my</i>	<i>le, plusieurs, mon</i>	<i>der, mehrere, mein</i>
Pronouns	<i>he, she, it</i>	<i>il, elle, lui</i>	<i>er, sie, ihm</i>
Prepositions	<i>to, of</i>	<i>vers, de</i>	<i>nach, von</i>
Conjunctions	<i>and, or</i>	<i>et, ou</i>	<i>und, oder</i>
Auxiliaries and modals	<i>be, have, will, would</i>	<i>être, avoir, pouvoir</i>	<i>sein, haben, können</i>

**Table 12.2** Open class categories

Part of speech	English	French	German
Nouns	<i>name, Frank</i>	<i>nom, François</i>	<i>Name, Franz</i>
Adjectives	<i>big, good</i>	<i>grand, bon</i>	<i>groß, gut</i>
Verbs	<i>to swim</i>	<i>nager</i>	<i>schwimmen</i>
Adverbs	<i>rather, very, only</i>	<i>plutôt, très, uniquement</i>	<i>fast, nur, sehr, endlich</i>

This is not new. To manage the nonexistence of articles in Latin, Latin grammarians tried to get the Greek article into the Latin pronoun category.

The definition of the parts of speech is sometimes arbitrary and has been a matter of debate. From Dionysius Thrax, tradition has defined the parts of speech using morphological and grammatical properties. We shall adopt essentially this viewpoint here. However, words of a certain part of speech share semantic properties, and some grammars contain statements like a noun denotes a thing and a verb an action.

Parts of speech can be clustered into two main classes: the **closed class** and the **open class**. Closed class words are relatively stable over time and have a functional role. They include words such as articles, like English *the*, French *le*, or German *der*, which change very slowly. Among the closed class, there are the determiners, the pronouns, the prepositions, the conjunctions, and the auxiliary and modal verbs (Table 12.1).

Open class words form the bulk of a vocabulary. They appear or disappear with the evolution of the language. If a new word is created, say a *hedgedog*, a cross between a hedgehog and a Yorkshire terrier, it will belong to an open class category: here a noun. The main categories of the open class are the nouns, the adjectives, the verbs, and the adverbs (Table 12.2). We can add interjection to this list. Interjections are words such as *ouch, ha, oh*, and so on, that express sudden surprise, pain, or pleasure.

### 12.1.2 Grammatical Features

Basic categories can be further refined, that is **subcategorized**. Nouns, for instance, can be split into singular nouns and plural nouns. In French and German, nouns

can also be split according to their gender: masculine and feminine for French, and masculine, feminine, and neuter for German.

Genders do not correspond in these languages and can shape different visions of the world. Sun is a masculine entity in French—*le soleil*—and a feminine one in German—*die Sonne*. In contrast, moon is a feminine entity in French—*la lune*—and a masculine one in German—*der Mond*.

Additional properties that can further specify main categories are often called the **grammatical features**. Grammatical features vary among European languages and include notably the number, gender, person, case, and tense. Each feature has a set of possible values; for instance, the number can be singular or plural.

Grammatical features are different according to their parts of speech. In English, a verb has a tense, a noun has a number, and an adjective has neither tense nor number. In French and German, adjectives have a number but no tense. The feature list of a word defines its part of speech together with its role in the sentence.

### **12.1.3 Two Significant Parts of Speech: The Noun and the Verb**

#### **The Noun**

Nouns are divided into proper and common nouns. Proper nouns are names of persons, people, countries, companies, and trademarks, such as *England*, *Robert*, *Citroën*. Common nouns are the rest of the nouns. Common nouns are often used to qualify persons, things, and ideas.

A noun definition referring to semantics is a disputable approximation, however. More surely, nouns have certain grammatical features, namely the number, gender, and case (Table 12.3). A noun group is marked with these features, and other words of the group, that is, determiners, adjectives, must agree with the features they share.

While number and gender are probably obvious, case might be a bit obscure for those who do know languages such as Latin, Russian, or German. Case is a function marker that inflects words such as nouns or adjectives. In German for example, there are four cases: nominative, accusative, genitive, and dative. The nominative case

**Table 12.3** Features of common nouns

Features\Values	English	French	German
Number	Singular, plural	Singular, plural	Singular, plural
	<i>Waiter/waiters, book/books</i>	<i>Serveur/serveurs, livre/livres</i>	<i>Buch/Bücher</i>
Gender		Masculine, feminine	Masculine, feminine, neuter
		<i>Serveur/table</i>	<i>Ober/Gabel/Tuch</i>
Case			Nominative, accusative, genitive, dative
			<i>Junge/Jungen/Jungen/Jungen</i>

corresponds to the subject function, the accusative case to the direct object function, and the dative case to the indirect object function. Genitive denotes a possession relation. These cases are still marked in English and French for pronouns.

In addition to these features, the English language makes a distinction between nouns that can have a plural: count nouns, and nouns that cannot: mass nouns. *Milk*, *water*, *air* are examples of mass nouns.

## Verbs

Semantically, verbs often describe an action, an event, a state, etc. More positively, and as for the nouns, verbs in European languages are marked by their morphology. This morphology, that is the form variation depending on the grammatical features, is quite elaborate in a language like French, notably due to the tense system. Verbs can be basically classified into three main types: auxiliaries, modals, and main verbs.

Auxiliaries are helper verbs such as *be* and *have* that enable us to build some of the main verb tenses. Modal verbs are verbs immediately followed by another verb in the infinitive. They usually indicate a modality, a possibility. Modal verbs are more specific to English and German. In French, semiauxiliaries correspond to a similar category.

Main verbs are all the other verbs. Traditionally, main verbs are categorized according to their complement's function:

- Copula or link verb—verbs linking a subject to an (adjective) complement. Copulas include **verbs of being** such as *be*, *être*, *sein* when not used as auxiliaries, and other verbs such as *seem*, *sembler*, *scheinen*.
- Intransitive—verbs taking no object.
- Transitive—verbs taking an object.
- Ditransitive—verbs taking two objects.

Verbs have more features than other parts of speech. First, the verb group shares certain features of the noun (Table 12.4). These features must agree with corresponding ones of the verb's subject.

Verbs have also specific features, namely the tense, the mode, and the voice:

**Tense** locates the verb, and the sentence, in time. Tense systems are elaborate in English, French, and German, and do not correspond. Tenses are constructed using form variations or auxiliaries. Tenses are a source of significant form variation in French;

**Mood** enables the speaker to present or to conceive of the action in various ways; **Voice** characterizes the sequence of syntactic groups. Active voice corresponds to the “subject, verb, object” sequence. The reverse sequence corresponds to the passive voice. This voice is possible only for transitive verbs. Some constructions in French and German use a reflexive pronoun. They correspond to the pronominal voice.

**Table 12.4** Features common to verbs and nouns

Features\Values	English	French	German
Person	1, 2, and 3	1, 2, and 3	1, 2, and 3
	<i>I am</i>	<i>je suis</i>	<i>ich bin</i>
	<i>You are</i>	<i>tu es</i>	<i>du bist</i>
	<i>She is</i>	<i>elle est</i>	<i>sie ist</i>
Number	Singular, plural	Singular, plural	Singular, plural
	<i>I am/we are</i>	<i>je suis/nous sommes</i>	<i>ich bin/wir sind</i>
	<i>She eats/they eat</i>	<i>elle mange/elles mangent</i>	<i>sie ißt/sie essen</i>
Gender		masculine, feminine	
	–	<i>il est mangé/elle est mangée</i>	–

## 12.2 Standardized Part-of-Speech Tagsets and Grammatical Features

While basic parts of speech are relatively well defined: determiners, nouns, pronouns, adjectives, verbs, auxiliaries, adverbs, conjunctions, and prepositions, there is a debate on how to standardize them for a computational analysis. One issue is the level of detail. Some tagsets feature a dozen tags, some over a hundred. Another issue that is linked to the latter is that of subcategories. How many classes for verbs? Only one, or should we create auxiliaries, modal, gerund, intransitive, transitive verbs, etc.?

The debate becomes even more complicated when we consider multiple languages. In French and German, the main parts of speech can be divided into subclasses depending on their gender, case, and number. In English, these divisions are useless. Although it is often possible to map tagsets from one language to another, there is no indisputable universal scheme, even within the same language.

Fortunately, with the collection and annotation of multilingual corpora, practical standards have emerged, although the discussion is not over. We will examine two annotation schemes: The Universal Part-of-Speech Tagset (UPOS) (Petrov et al. 2012) for parts of speech and an extension of MULTTEXT (Ide and Véronis 1995; Monachini and Calzolari 1996) for grammatical features.

### 12.2.1 Multilingual Part-of-Speech Tags

Building a multilingual tagset imposes the condition of having a set of common classes, which enables a comparison between languages. These classes correspond to traditional parts of speech and gather a relatively large consensus among European languages. However, they are not sufficiently accurate for any language in particular. Dermatas and Kokkinakis (1995) retained the traditional parts of speech

**Table 12.5** Parts of speech and grammatical features

Main parts of speech	Features (subcategories)
Adjective, noun, pronoun	Regular base comparative superlative interrogative person number case
Adverb	Regular base comparative superlative interrogative
Article, determiner, preposition	Person case number
Verb	Tense voice mood person number case

**Table 12.6** UPOS part-of-speech tagset

Part of speech	Universal POS tagset
Noun	NOUN
Proper noun	PNOUN
Verb	VERB
Auxiliary verb	AUX
Adjective	ADJ
Determiner (including article)	DET
Numerical	NUM
Adverb	ADV
Pronoun	PRON
Preposition (adposition and postposition)	ADP
Coordinating conjunction	CCONJ
Subordinating conjunction	SCONJ
Interjection	INTJ
Particle	PART
Other (foreign words, etc.)	X
Symbol (nonpunctuation)	SYM
Punctuation	PUNCT

to tag texts in seven European languages using statistical methods. They also added features (subcategories) specific to each language (Table 12.5).

MULTEXT (Ide and Véronis 1995; Monachini and Calzolari 1996) was a multinational initiative that aimed at providing an annotation scheme for all the Western and Eastern European languages. For the parts of speech, MULTEXT merely perpetuated the traditional categories and assigned them a code. The universal part-of-speech tagset (UPOS) (Petrov et al. 2012) is almost identical, but includes mappings with other tagsets used in older corpora. This ensured its popularity and UPOS is now widely adopted; see Table 12.6.

**Table 12.7** Features (attributes) and values for nouns

Position	Attribute	Value	Code
1	Type	Common	c
		Proper	p
2	Gender	Masculine	m
		Feminine	f
		Neuter	n
3	Number	Singular	s
		Plural	p
4	Case	Nominative	n
		Genitive	g
		Dative	d
		Accusative	a

## 12.2.2 Multilingual Grammatical Features

MULTEXT complemented the parts of speech with a set of grammatical features, which they called attributes. Attributes enable us to subcategorize words and reconcile specific features of different European languages. Tables 12.7 and 12.8 show attributes for nouns and verbs.

MULTEXT attributes concern only the morpho-syntactic layer and represent a superset of what is needed by all the languages. Some attributes may not be relevant for a specific language. For instance, English nouns have no gender, and French ones have no case. In addition, applications may not make use of some of the attributes even if they are part of the language. Tense, for instance, may be useless for some applications.

A part-of-speech tag is a string where the first character is the main class of the word to annotate and then a sequence of attribute values. Attribute positions correspond to their rank in the table, such as those defined in Tables 12.7 and 12.8 for nouns and verbs. When an attribute is not applicable, it is replaced by a dash (-). An English noun could receive the tag:

```
N[type=common number=singular] Nc-s-
```

a French one:

```
N[type=common gender=masculine number=singular] Ncms-
```

and a German one:

```
N[type=common gender=neuter number=singular  
case=nominative] Ncnsn
```

A user can extend the coding scheme and add attributes if the application requires it. A noun could be tagged with some semantic features such as country names, currencies, etc.

**Table 12.8** Attributes (features) and values for verbs

Position	Attribute	Value	Code
1	Type	Main	m
		Auxiliary	a
		Modal	o
2	Mood/form	Indicative	i
		Subjunctive	s
		Imperative	m
		Conditional	c
		Infinitive	i
		Participle	p
		Gerund	g
		Supine	s
3	Tense	Base	b
		Present	p
		Imperfect	i
		Future	f
		Past	s
4	Person	First	1
		Second	2
		Third	3
5	Number	Singular	s
		Plural	p
6	Gender	Masculine	m
		Feminine	f
		Neuter	n

Finally, it is easy to see that the sequence of attributes, such as Nc-s-, is just a duplicate of the list of keys and values, N[type=common number=singular]. In addition, this sequence needs to be ordered. We can remove the redundancy and just use the UPOS codes and a set of key/value pairs separated by vertical bars. We can encode the three examples above as:

```
NOUN number=singular
NOUN gender=masculine|number=singular
NOUN gender=neuter|number=singular|case=nominative
```

where we write the pairs in any order. When associated with UPOS, we do not need a noun type as the UPOS noun is a common noun.

This format for grammatical features is adopted by CoNLL-U and the Universal Dependencies corpora. It has become a sort of *de facto* standard; see the two next sections.

## 12.3 The CoNLL Format

The Conference on Natural Language Learning (CoNLL) is an annual conference dedicated to statistical and machine-learning techniques in language analysis. In addition to the classical contributions in the form of articles found in scientific conferences, CoNLL organizes a “shared task” to evaluate language processing systems on a specific problem. As hinted by the conference name, the competing systems should use machine-learning techniques, essentially supervised learning. The participants are given a training set to train their models and are evaluated on a test set. CoNLL makes these datasets available in a column-based format as shown in Tables 12.9 and 12.11.

The CoNLL format has become very popular as it can be used to annotate any layer of a linguistic analysis: part-of-speech tagging, morphological parsing, dependency parsing, semantic parsing, coreference, etc. One of its main characteristics is that it has one word per line with the word properties and annotation shown on the same line in separate columns. The content of a given column may vary depending on the task as well as the total number of columns.

**Table 12.9** Annotation of the Spanish sentence: *La reestructuración de los otros bancos checos se está acompañando por la reducción del personal* ‘The restructuring of Czech banks is accompanied by the reduction of personnel’ (Palomar et al. 2004) using the CoNLL-U format

ID	FORM	LEMMA	UPOS	FEATS
1	La	el	DET	Definite=Def Gender=Fem  Number=Sing PronType=Art
2	reestructuración	reestructuración	NOUN	Gender=Fem Number=Sing
3	de	de	ADP	AdpType=Prep
4	los	el	DET	Definite=Def Gender=Masc  Number=Plur PronType=Art
5	otros	otro	DET	Gender=Mascl Number=Plur  PronType=Ind
6	bancos	banco	NOUN	Gender=Mascl Number=Plur
7	chechos	checo	ADJ	Gender=Mascl Number=Plur
8	se	se	PRON	Case=Acc Person=3 PrepCase=Npr  PronType=Prs Reflex=Yes
9	está	estar	AUX	Mood=Ind Number=Sing Person=3  Tense=Pres VerbForm=Fin
10	acompañando	acompañar	VERB	VerbForm=Ger
11	por	por	ADP	AdpType=Prepron
12	la	el	DET	Definite=Def Gender=Fem  Number=Sing PronType=Art
13	reducción	reducción	NOUN	Gender=Fem Number=Sing
14	del	del	ADP	AdpType=Prepron
15	personal	personal	NOUN	Gender=Mascl Number=Sing
16	.	.	PUNCT	PunctType=Peri

The lexical and part-of-speech annotation of CoNLL gradually evolved across the years into a format equivalent to that of MULTEXT. CoNLL-U (de Marneffe et al. 2021) is the latest version and Table 12.9 exemplifies it with the Spanish sentence (Palomar et al. 2004):

La reestructuración de los otros bancos checos se está acompañando por la reducción del personal.

‘The restructuring of Czech banks is accompanied by the reduction of personnel’.

where

- The ID column is the word index in the sentence;
- The FORM column corresponds to the word;
- The LEMMA column contains the lemma and the phrase *los otros bancos* starting at index 4 is lemmatized as *el otro banco*;
- The UPOS column corresponds to the part of speech: *los* as well as *otros* are determiners; *bancos* is a noun;
- Finally, the FEATS column corresponds to the grammatical features that are listed as an unordered set separated by vertical bars. The word *bancos* ‘banks’ has a masculine gender (**Gender=Masc**) and a plural number (**Number=Plur**).

The columns are delimited by a tabulation character, and the sentences by a blank line. Each sentence is preceded by comments starting with a # character giving the sentence identifier as well as a raw text version of it:

```
# sent_id = 3LB-CAST-104_C-5-s10
# text = La reestructuración de los otros bancos checos se \
          está acompañando por la reducción del personal.
# orig_file_sentence 001#29
```

Tables 12.10 and 12.11 show a similar annotation with respectively a sentence in English from the EWT corpus and a sentence from the French FTB corpus (Abeillé et al. 2003; Abeillé and Clément 2003) converted to the CoNLL-U format by Silveira et al. (2014) and Candito et al. (2009).

## 12.4 A CoNLL Reader in Python

The Universal Dependencies (UD) repository (de Marneffe et al. 2021) consists of more than 200 annotated corpora in more than 100 languages. They all adopt the CoNLL-U format that we described in Sects. 12.2 and 12.3 for the parts of speech as well as for the grammatical features. As it is quite large and multilingual, it is a valuable source of training data to build morphological parsers and part-of-speech taggers. We describe here a Python reader to load a corpus and extract the forms, lemmas, and features.

**Table 12.10** Annotation of the English sentence: *Or you can visit temples or shrines in Okinawa.* from the EWT corpus following the CoNLL-U format (Silveira et al. 2014)

ID	FORM	LEMMA	UPOS	FEATS
1	Or	or	CCONJ	_
2	you	you	PRON	Case=Nom Person=2 PronType=Prs
3	can	can	AUX	VerbForm=Fin
4	visit	visit	VERB	VerbForm=Inf
5	temples	temple	NOUN	Number=Plur
6	or	or	CCONJ	_
7	shrines	shrine	NOUN	Number=Plur
8	in	in	ADP	_
9	Okinawa	Okinawa	PROPN	Number=Sing
10	.	.	PUNCT	_

**Table 12.11** Annotation of the French sentence: *À cette époque, on avait dénombré cent quarante candidats* ‘At that time, we had counted one hundred and forty candidates’ (Abeillé et al. 2003; Abeillé and Clément 2003) following the CoNLL-U format

ID	FORM	LEMMA	UPOS	FEATS
1	À	à	ADP	_
2	cette	ce	DET	Gender=Fem Number=Sing PronType=Dem
3	époque	époque	NOUN	Gender=Fem Number=Sing
4	,	,	PUNCT	_
5	on	il	PRON	Gender=Masc Number=Sing Person=3
6	avait	avoir	AUX	Mood=Ind Number=Sing Person=3  Tense=Impl VerbForm=Fin
7	dénombré	dénombrer	VERB	Gender=Masc Number=Sing  Tense=Past VerbForm=Part
8	cent	cent	NUM	NumType=Card
9	quarante	quarante	NUM	NumType=Card
10	candidats	candidat	NOUN	Gender=Masc Number=Plur
11	.	.	PUNCT	_

The UD corpora are available from GitHub<sup>1</sup> and we can collect them using the techniques described in Sect. 2.14. Then to extract the fields of a CoNLL corpus, we need to parse its structure. We give here an example program consisting of two classes for the CoNLL-U format. We can easily adapt it to the slight format changes used by the different CoNLL tasks. The first class, `Token`, models the row of a CoNLL corpus: An annotated word. We just create a subclass of a dictionary:

```
class Token(dict):
    pass
```

<sup>1</sup> <https://github.com/UniversalDependencies>.

We create a token object by passing a dictionary representing the row and all its columns, as for instance for the first row in Table 12.9:

```
tok = Token({'ID': '1', 'FORM': 'La', 'LEMMA': 'el',
            'UPOS': 'DET', 'FEATS':
            'Definite=Def|Gender=Fem|Number=Sing|PronType=Art'})
```

The second class, `CoNLLDictorizer`, transforms the corpus content into a list of sentences, where each sentence is a list of `Tokens`. We follow the transformer structure of scikit-learn to write this class to integrate it more easily with other modules of this library. A transformer has two methods, `fit()` to learn some parameters, and `transform()` to apply the transformation. Here, we will only use the latter. When we create a `corpus` object, we pass the names of the columns, the sentence separator, and the column separator, and we store them in the object. Then `transform()` splits the corpus into sentences, each sentence is split into rows, and each row is split into columns. We create a `Token` of each row.

```
class CoNLLDictorizer:

    def __init__(self, column_names,
                 sent_sep='\n\n',
                 col_sep='\t+'):
        self.column_names = column_names
        self.sent_sep = sent_sep
        self.col_sep = col_sep

    def fit(self):
        pass

    def transform(self, corpus):
        corpus = corpus.strip()
        sentences = re.split(self.sent_sep, corpus)
        return list(map(self._split_in_words, sentences))

    def fit_transform(self, corpus):
        return self.transform(corpus)

    def _split_in_words(self, sentence):
        rows = re.split('\n', sentence)
        rows = [row for row in rows if row[0] != '#']
        return [Token(dict(zip(self.column_names,
                               re.split(self.col_sep, row)))) for row in rows]
```

To read the corpus, we need the column names. For CoNLL-U, these are:

```
col_names = ['ID', 'FORM', 'LEMMA', 'UPOS', 'XPOS', 'FEATS',
            'HEAD', 'DEPREL', 'HEAD', 'DEPS', 'MISC']
```

We create a `CoNLLDictorizer` object with these names and we call `transform()` with the content of a CoNLL file, here `train_sentences`:

```
conll_dict = CoNLLDictorizer(column_names)
train_dict = conll_dict.transform(train_sentences)
```

This method returns `train_dict`, a list of dictionaries. Each list contains one sentence and each dictionary contains one word with the column names as keys. Printing the form, lemma, part of speech, and features of the four first words of the sentence in Table 12.10 results in:

```
[{'ID': '1', 'FORM': 'Or', 'LEMMA': 'or', 'UPOS': 'CCONJ',
 'FEATS': '_'},
 {'ID': '2', 'FORM': 'you', 'LEMMA': 'you', 'UPOS': 'PRON',
 'FEATS': 'Case=Nom|Person=2|PronType=Prs'},
 {'ID': '3', 'FORM': 'can', 'LEMMA': 'can', 'UPOS': 'AUX',
 'FEATS': 'VerbForm=Fin'},
 {'ID': '4', 'FORM': 'visit', 'LEMMA': 'visit', 'UPOS': 'VERB',
 'FEATS': 'VerbForm=Inf'}
 ...]
```

## 12.5 Lexicons

A lexicon is a list of words, and in this context, lexical entries are also called the **lexemes**. Lexicons often cover a particular domain. Some focus on a whole language, like English, French, or German, while some specialize in specific areas such as proper names, technology, science, and finance. In some applications, lexicons try to be as exhaustive as is humanly possible. This is the case of internet crawlers, which index all the words of all the web pages they can find. Computerized lexicons are now embedded in many popular applications such as in spelling checkers, thesauruses, or definition dictionaries of word processors. They are also often the first building block of most language processing programs.

Several options can be taken when building a computerized lexicon. They range from a collection of words—a word list—to words carefully annotated with their pronunciation, morphology, and syntactic and semantic labels. Words can also be related together using semantic relationships and definitions.

A key point in lexicon building is that many words are ambiguous both syntactically and semantically. Therefore, each word may have as many entries as it has syntactic or semantic readings. Table 12.12 shows words that have two or more parts of speech and senses.

Computerized lexicons are now available from industry and from sources on the Internet in English and many other languages. Most notable ones in English include word lists derived from the *Longman Dictionary of Contemporary English* (Procter 1978) and the *Oxford Advanced Learner's Dictionary* (Hornby 1974). Table 12.13 shows the first lines of letter A of an electronic version of the OALD.

**Table 12.12** Word ambiguity

	English	French	German
Part of speech	<i>can</i> modal	<i>le</i> article	<i>der</i> article
	<i>can</i> noun	<i>le</i> pronoun	<i>der</i> pronoun
Semantic	<i>great</i> big	<i>grand</i> big	<i>groß</i> big
	<i>great</i> notable	<i>grand</i> notable	<i>groß</i> notable

**Table 12.13** The first lines of the *Oxford Advanced Learner's Dictionary*

Word	Pronunciation	Syntactic tag	Syllable count or verb pattern (for verbs)
a	@	S-*	1
a	EI	Ki\$	1
a fortiori	eI ,fOtI'Oral	Pu\$	5
a posteriori	eI ,p0sterI'Oral	OA\$,Pu\$	6
a priori	eI ,pral'Oral	OA\$, Pu\$	4
a's	Eiz	Kj\$	1
ab initio	&b I'nISI@U	Pu\$	5
abaci	'&b@sal	Kj\$	3
aback	@'b&k	Pu%	2
abacus	'&b@k@s	K7%	3
abacuses	'&b@k@sIz	Kj%	4
abaft	@'bAft	Pu\$,T-\$	2
abandon	@'b&nd@n	H0%,L@%	36A,14
abandoned	@'b&nd@nd	Hc%,Hd%,OA%	36A,14
abandoning	@'b&nd@nIN	Hb%	46A,14
abandonment	@'b&nd@nm@nt	L@%	4
abandons	@'b&nd@nz	Ha%	36A,14
abase	@'beIs	H2%	26B
abased	@'beIst	Hc%,Hd%	26B
abasement	@'beIsm@nt	L@%	3

BDLex—standing for *Base de Données Lexicale*—is an early example of a simple French lexicon (Pérennou and de Calmès 1987). BDLex features a list of words in a lemmatized form together with their part of speech and a syntactic type (Table 12.14).

As we can see from the previous examples, the lexicon formats are quite disparate. The XML Text Encoding Initiative (XML-TEI) (TEI Consortium 2023), based on XML (see Sect. 4.4), is a standardization attempt to facilitate the exchange and reuse of annotated textual data. The XML-TEI defines structures for different categories of text with a focus on digital humanities. It notably features specific guidelines to encode mono and multilingual dictionaries with fields for the headwords, pronunciation, grammatical information, etymology, senses, and definitions. Notable examples of adoption of the XML-TEI standard include the digital transcription of a historic edition of the French *Petit Larousse illustré*

**Table 12.14** An excerpt from BDLex. Digits encode accents on letters. The syntactical tags of the verbs correspond to their conjugation type taken from the *Bescherelle* reference

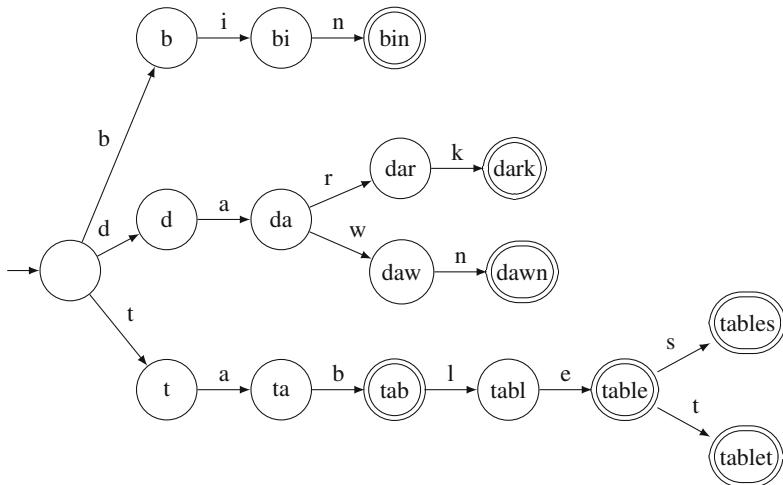
Entry	Part of speech	Lemma	Syntactic tag
a2	Prep	a2	Prep_00_00;
abaisser	Verbe	abaisser	Verbe_01_060_**;
abandon	Nom	abandon	Nom_Mn_01;
abandonner	Verbe	abandonner	Verbe_01_060_**;
abattre	Verbe	abattre	Verbe_01_550_**;
abbe1	Nom	abbe1	Nom_gn_90;
abdiquer	Verbe	abdiquer	Verbe_01_060_**;
abeille	Nom	abeille	Nom_Fn_81;
abi3mer	Verbe	abi3mer	Verbe_01_060_**;
abolition	Nom	abolition	Nom_Fn_81;
abondance	Nom	abondance	Nom_Fn_81;
abondant	Adj	abondant	Adj_gn_01;
abonnement	Nom	abonnement	Nom_Mn_01;
abord	Nom	abord	Nom_Mn_01;
aborder	Verbe	aborder	Verbe_01_060_**;
aboutir	Verbe	aboutir	Verbe_00_190_**;
aboyer	Verbe	aboyer	Verbe_01_170_**;
abrelger	Verbe	abrelger	Verbe_01_140_**;
abrelviation	Nom	abrelviation	Nom_Fn_81;
abri	Nom	abri	Nom_Mn_01;
abriter	Verbe	abriter	Verbe_01_060_**;

dictionary from 1905 (Bohbot et al. 2018) and Diderot’s *Encyclopédie* from 1751–1772 (Guilbaud 2017).

### 12.5.1 Encoding a Dictionary

Letter trees (de la Briandais 1959) or tries (pronounce try ees) are useful data structures to store large lexicons and to search words quickly. The idea behind a trie is to store the words as trees of characters and to share branches as far as the letters of two words are identical. Tries can be seen as finite-state automata, and Fig. 12.1 shows a graphical representation of a trie encoding the words *bin*, *dark*, *dawn*, *tab*, *table*, *tables*, and *tablet*.

In Python, we can represent this trie as embedded lists, where each branch is a list. The first element of a branch is the root letter: the first letter of all the subwords that correspond to the branch. The leaves of the trie are the lexical entries, here the words themselves. Of course, these entries could contain more information, such as the part of speech, the pronunciation, etc.



**Fig. 12.1** A letter tree encoding the words *tab*, *table*, *tablet*, and *tables*

```

[
  ['b', ['i', ['n', 'bin']]],
  ['d', ['a', ['r', ['k', 'dark']],
         ['w', ['n', 'dawn']]],
  ['t', ['a', ['b', 'tab',
              ['l', ['e', 'table',
                     ['s', 'tables'],
                     ['t', 'tablet']]]]]]
]
```

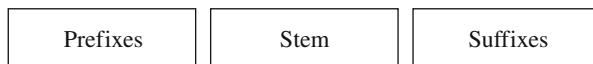
## 12.6 Morphology

### 12.6.1 Morphemes

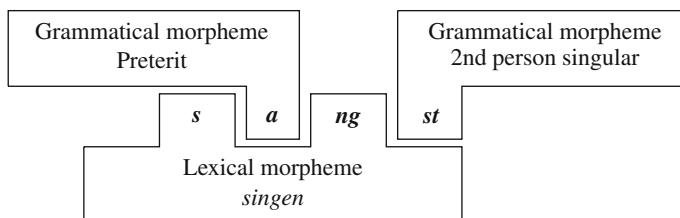
From a morphological viewpoint, a language is a set of morphemes divided into **lexical** and **grammatical** morphemes. Lexical morphemes correspond to the word stems and form the bulk of the vocabulary. Grammatical morphemes include grammatical words and the affixes. In European languages, words are made of one or more morphemes (Table 12.15). The affixes are concatenated to the stem (bold): before it—the prefixes (underlined)—and after it—the suffixes (double underlined). When a prefix and a suffix surrounding the stem are bound together, it is called a circumfix, as with the (ge-, -t) or (ge-, -en) pairs in the German part participle (wavy underlines).

**Table 12.15** Morpheme decomposition. We replaced the stems with the corresponding lemmas

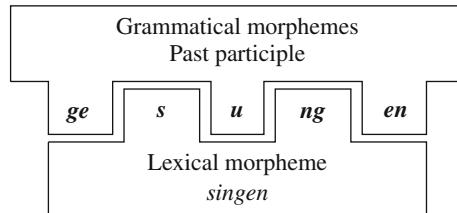
	Word	Morpheme decomposition
English	<i>disentangling</i>	<u>dis+en+tangle+ing</u>
	<i>rewritten</i>	<u>re+write+en</u>
French	<i>désembrouillé</i>	<u>dé+em+brouiller+é</u>
	<i>récrite</i>	<u>re+écrire+te</u>
German	<i>entwirrend</i>	<u>ent+wirren+end</u>
	<i>wiedergeschrieben</i>	<u>wieder+ge+schreiben+en</u>



**Fig. 12.2** Concatenative morphology where prefixes and suffixes are concatenated to the stem



**Fig. 12.3** Embedding of the stem into the grammatical morphemes in the German verb *sangst* (second-person preterit of *singen*). After Simone (2007, p. 144)



**Fig. 12.4** Embedding of the stem into the grammatical morphemes in the German verb *gesungen* (past participle of *singen*). After Simone (2007, p. 144)

Affixing grammatical morphemes to the stem is a general property of most European languages, which is **concatenative morphology** (Fig. 12.2). Although there are numerous exceptions, it enables us to analyze the structure of most words.

Concatenative morphology is not universal, however. The Semitic languages, like Arabic or Hebrew, for instance, have a **templetic morphology** that interweaves the grammatical morphemes to the stem. There are also examples of nonconcatenative patterns in European languages like in irregular verbs of German. The verb *singen* ‘sing’ has the forms *sangst* ‘you sang’ and *gesungen* ‘sung’ where the stem [s–ng] is embedded into the grammatical morphemes [–a–st] for the second-person preterit (Fig. 12.3) and [ge–u–en] for the past participle (Fig. 12.4).

**Table 12.16** Plural morphs

	Plural of nouns	Morpheme decomposition
English	<i>hedgehogs</i>	<i>hedgehog+s</i>
	<i>churches</i>	<i>church+es</i>
	<i>sheep</i>	<i>sheep+Ø</i>
French	<i>hérissons</i>	<i>hérisson+s</i>
	<i>chevaux</i>	<i>cheval+ux</i>
German	<i>Gründe</i>	<i>Grund+(‘)e</i>
	<i>Hände</i>	<i>Hand+(‘)e</i>
	<i>Igel</i>	<i>Igel+Ø</i>

## 12.6.2 Morphs

Grammatical morphemes represent syntactic or semantic functions whose realizations in words are called **morphs**. Using an object-oriented terminology, morphemes would be the classes, while morphs would be the objects. The **allo-morphs** correspond to the set of all the morphs in a morpheme class.

The plural morpheme of English and French nouns is generally realized with an *s* suffix—an *s* added at the end of the noun. It can also be *es* or nothing ( $\emptyset$ ) in English and *ux* in French. In German, the plural morpheme can take several shapes, such as suffixes *e*, *en*, *er*, *s*, or an umlaut on the first vowel of the word (Table 12.16):

- In English, suffixes *-s*, *-es*, etc.
- In French, *-s*, *-ux*, etc.
- In German, an umlaut on the first vowel and the *-e* suffix, or simply the *-e* suffix.

Plurals also offer exceptions. Many of the exceptions, such as *mouse* and *mice*, are not predictable and have to be listed in the lexicon.

## 12.6.3 Inflection and Derivation

### Some Definitions

Morphology can be classified into **inflection**, the form variation of a word according to syntactic features such as gender, number, person, tense, etc., and **derivation**, the creation of a new word—a new meaning—by concatenating a word with a specific affix. A last form of construction is the **composition (compounding)** of two words to give a new one, for instance, *part of speech*, *can opener*, *pomme de terre*. Composition is more obvious in German, where such new words are not separated with a space, for example, *Führerschein*. In English and French, some words are formed in this way, such as *bedroom*, or are separated with a hyphen, *centre-ville*. However, the exact determination of other compounded words—separated with a space—can be quite tricky.

**Table 12.17** Verb inflection with past participle

	English	French	German
Base form	<i>work</i>	<i>travailler, chanter</i>	<i>arbeiten</i>
	<i>sing</i>	<i>paraître</i>	<i>singen</i>
Past participle (regular)	<i>worked</i>	<i>travaillé, chanté</i>	<i>gearbeitet</i>
Past participle (exception)	<i>sung</i>	<i>paru</i>	<i>gesungen</i>

## Inflection

Inflection corresponds to the application of a grammatical feature to a word, such as putting a noun into the plural or a verb into the past participle (Table 12.17). It is also governed by its context in the sentence; for instance, the word is bound to agree in number with some of its neighbors.

Inflection is relatively predictable—regular—depending on the language. Given a lemma, its part of speech, and a set of grammatical features, it is possible to construct a word form using rules, for instance, gender, plural, or conjugation rules. The past participle of regular English, French, and German verbs can be respectively formed with an *ed* suffix, an *é* suffix, and the *ge* prefix and the *t* suffix. Morphology also includes frequent exceptions that can sometimes also be described by rules.

Inflectional systems are similar in European languages but show differences according to the syntactic features. In English, French, and German, nouns are inflected with plurals and are consequently decorated with a specific suffix. However, in French and other Romance languages, verbs are inflected with future. Verb *chanterons* is made of two morphs: *chant* ‘sing’ and *-erons*. The first one is the stem (root) of *chanter*, and the second one is a suffix indicating the future tense, the first person, and the plural number. In English and German, this tense is rendered with an auxiliary: *we shall sing* or *wir werden singen*.

## Derivation

Derivation is linked to lexical semantics and involves another set of affixes (Table 12.18). Most affixes can only be attached to a specific lexical category (part of speech) of words: some to nouns, others to verbs, etc. Some affixes leave the derived word in the same category, while some others entail a change of category. For instance, some affixes transform adjectives into adverbs, nouns into adjectives, and verbs into nouns (Table 12.19). Derivation rules can be combined and are sometimes complex. For instance, the word *disentangling* features two prefixes: *dis-* and *en-*, and a suffix *-ing*.

Some semantic features of words, such as the contrary or the possibility, can be roughly associated to affixes, and so word meaning can be altered using them (Table 12.20). However, derivation is very irregular. Many words cannot be

**Table 12.18** Derivational affixes

	English	French	German
Prefixes	<i>foresee, unpleasant</i>	<i>prévoir, déplaisant</i>	<i>vorhersehen, unangenehm</i>
Suffixes	<i>manageable, rigorous</i>	<i>gérable, rigoureux</i>	<i>vorsichtig, streitbar</i>

**Table 12.19** Derivation related to part of speech

	Adjectives	Adverbs	Nouns	Adjectives	Verbs	Nouns
English	<i>recent</i>	<i>recently</i>	<i>air</i>	<i>aerial</i>	<i>compute</i>	<i>computation</i>
	<i>frank</i>	<i>frankly</i>	<i>base</i>	<i>basic</i>		
French	<i>récent</i>	<i>récemment</i>	<i>lune</i>	<i>lunaire</i>	<i>calculer</i>	<i>calcul</i>
	<i>franc</i>	<i>franchement</i>	<i>air</i>	<i>aérien</i>		
German	<i>glücklich</i>	<i>glücklicherweise</i>	<i>Luft</i>	<i>luftig</i>	<i>rechnen</i>	<i>Rechnung</i>
	<i>möglich</i>	<i>möglicherweise</i>	<i>Grund</i>	<i>gründlich</i>		

**Table 12.20** Word derivation

	Word	Contrary	Possibility
English	<i>pleasant do</i>	<i>unpleasant undo</i>	* <i>pleasable doable</i>
French	<i>plaisant faire</i>	<i>déplaisant défaire</i>	* <i>plaisable faisable</i>
German	<i>angenehm tun</i>	<i>unangenehm *untun</i>	* <i>angenehmbar tunlichst</i>

generated as simply, because the word does not exist or sounds weird. In addition, some affixes cannot be mapped to clear semantic features.

Compounding is a feature of German, Dutch, and the Scandinavian languages. It resembles the English noun sequences with the difference that nouns are not separated with a white space.

## Morphological Processing

Morphological processing includes parsing and generation (Table 12.21). Parsing consists in splitting an inflected, derived, or compounded word into morphemes; this process is also called a **lemmatization**. Lemmatization refers to transforming a word into its canonical dictionary form, for example, *retrieving* into *retrieve*, *recherchant* into *rechercher*, or *suchend* into *suchen*. Stemming consists of removing the suffix from the rest of the word. Taking the previous examples, this yields *retriev*, *recherch*, and *such*. Lemmatization and stemming are often mistaken. Conversely, generation consists of producing a word—a lexical form—from a set of morphemes.

In French, English, and German, derivation operates on open class words. In English and French, a word of this class consists of a stem preceded by zero or more derivational prefixes and followed by zero or more derivational suffixes. An inflectional suffix can be appended to the word. In German, a word consists of one or more stems preceded by zero or more derivational prefixes and followed by zero or more derivational suffixes. An inflectional prefix and an inflectional suffix can

**Table 12.21** Morphological generation and parsing

Generation →					
English		French		German	
<i>dog+s</i>	<i>dogs</i>	<i>chien+s</i>	<i>chiens</i>	<i>Hund+e</i>	<i>Hunde</i>
<i>work+ing</i>	<i>working</i>	<i>travailler+ant</i>	<i>travaillant</i>	<i>arbeiten+end</i>	<i>arbeitend</i>
<i>un+do</i>	<i>undo</i>	<i>dé+faire</i>	<i>défaire</i>		

← Parsing

**Table 12.22** Open class word morphology, where \* denotes zero or more elements and ? denotes an optional element

English and French	<b>prefix*</b> <b>stem</b> <b>suffix*</b> <b>inflection?</b>
German	<b>inflection?</b> <b>prefix*</b> <b>stem*</b> <b>suffix*</b> <b>inflection?</b>

**Table 12.23** Lemmatization ambiguities

	Words	Words in context	Lemmatization
English	<i>Run</i>	1. <i>A run in the forest</i> 2. <i>Sportsmen run every day</i>	1. <b>run</b> : noun singular 2. <b>run</b> : verb present third person plural
French	<i>Marche</i>	1. <i>Une marche dans la forêt</i> 2. <i>Il marche dans la cour</i>	1. <b>marche</b> : noun singular feminine 2. <b>marcher</b> : verb present third person singular
German	<i>Lauf</i>	1. <i>Der Lauf der Zeit</i> 2. <i>Lauf schnell!</i>	1. <b>Der Lauf</b> : noun, sing, masc 2. <b>laufen</b> : verb, imperative, singular

be appended to the word (Table 12.22). As we saw earlier, these rules are general principles of concatenative morphology that have exceptions.

## Ambiguity

Word lemmatization is often ambiguous. An isolated word can lead to several readings: several bases and morphemes, and in consequence several categories and features as exemplified in Table 12.23.

Lemmatization ambiguities are generally resolved using the word context in the sentence. Usually only one reading is syntactically or semantically possible, and others are not. The correct reading of a word's part of speech is determined considering the word's relations with the surrounding words and with the rest of the sentence. From a human perspective, this corresponds to determining the word's

function in the sentence. As we saw in the introduction, this process has been done by generations of pupils dating as far back as the schools of ancient Greece and the Roman Empire.

### 12.6.4 Language Differences

Paper lexicons do not include all the words of a language but only lemmas. Each lemma is fitted with a morphological class to relate it to a model of inflection or possible exceptions. A French verb will be given a class of conjugation or its exception pattern—one among a hundred. English or German verbs will be marked as regular or strong and in this latter case will be given their irregular forms. Then, a reader can apply morphological rules to produce all the lexical forms of the language.

Automatic morphological processing tries to mimic this human behavior. Nevertheless, it has not been so widely implemented in English as in other languages. Programmers have often preferred to pack all the English words into a single dictionary instead of implementing a parser to do the job. This strategy is possible for European languages because morphology is finite: there is a finite number of noun forms, adjective forms, or verb forms. It is clumsy, however, to extend it to languages other than English because it considerably inflates the size of dictionaries.

Statistics from Xerox (Table 12.24) show that techniques available for storing English words are very costly for many other languages. It is not a surprise that the most widespread morphological parser—KIMMO—was originally built for Finnish, one of the most inflection-rich languages. In addition, while English inflection is tractable by means of storing all the forms in a lexicon, it is often necessary to resort to a morphological parser to deal with forms such as: *computer*, *computerize*, *computerization*, *recomputerize* (Antworth 1994), which cannot all be foreseen by lexicographers.

**Table 12.24** Some language statistics from a Xerox promotional flyer

Language	Number of stems	Number of inflected forms	Lexicon size (kb)
English	55,000	240,000	200–300
French	50,000	5,700,000	200–300
German	50,000	350,000 infinite	450 (compounding)
Japanese	130,000	200	500 Suffixes
		20,000,000	Word forms
Spanish	40,000	3,000,000	200–300

## 12.7 Morphological Parsing

### 12.7.1 Two-Level Model of Morphology

Using a memory expensive method, lemmatization can be accomplished with a lexicon containing all the words with all their possible inflections. A dictionary lookup then yields the lemma of each word in a text. Although it has often been used for English, this method is not very efficient for many other languages. We now introduce the two-level model of Kimmo Koskenniemi (1983), which is universal and has been adopted by many morphological parsers.

The two-level morphology model enables us to link the **surface form** of a word—the word as it is actually in a text—to its **lexical** or **underlying form**—its sequence of morphemes. Karttunen (1983) did the first implementation of this model, which he named KIMMO. A later implementation—PC-KIMMO 2—was carried out by Antworth (1995) in C.

Table 12.25 shows examples of correspondence between surface forms and lexical forms. Morpheme boundaries in lexical forms are denoted by +.

In the two-level model, the mapping between the surface and lexical forms is synchronous. Both strings need to be aligned with a letter-for-letter correspondence. That is, the first letter of the first form is mapped to the first letter of the second form, and so on. To maintain the alignment, possible null symbols are inserted in either form and are denoted  $\varepsilon$  or 0, if the Greek letters are not available. They reflect a letter deletion or insertion. Table 12.26 shows aligned surface and lexical forms.

### 12.7.2 Interpreting the Morphs

Considering inflection only, it is easier to interpret the morphological information using grammatical features rather than morphs. Most morphological parsers represent the lexical form as a concatenation of the stem and its features instead of

**Table 12.25** Surface and lexical forms

Generation: Lexical to surface form →		
English	<i>dis+en+tangle+ed</i>	<i>disentangled</i>
	<i>happy+er</i>	<i>happier</i>
	<i>move+ed</i>	<i>moved</i>
French	<i>dés+em+brouiller+é</i>	<i>désembrouillé</i>
	<i>dé+chanter+erons</i>	<i>déchanterons</i>
German	<i>ent+wirren+end</i>	<i>entwirrend</i>
	<i>wieder+ge+schreiben+en</i>	<i>wiedergeschrieben</i>
Parsing: ← Surface to lexical form		

**Table 12.26** Correspondence between lexical and surface forms

English	dis+en+tangle+ed	happy+er	move+ed
	↑↓ ... dis0en0tangl00ed	↑↓ ... happi0er	↑↓ ... mov00ed
French	dé+chanter+erons	cheval+ux	cheviller+é
	↑↓ ... dé0chant000erons	↑↓ ... cheva00ux	↑↓ ... chevill000é
German	singen+st	Grund+“e	Igel+Ø
	↑↓ ... sing00st	↑↓ ... Gründ00e	↑↓ ... Igel00

Lexical:	d	i	s	e	n	t	a	n	g	l	e	+Verb	+PastBoth	+123sp
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↑	↑	↑
Surface:	d	i	s	e	n	t	a	n	g	l	0	0	e	d

Lexical:	h	a	p	p	y	+Adj	+Comp
	↓	↓	↓	↓	↓	↓	↓
Surface:	h	a	p	p	i	e	r

Lexical:	G	r	u	n	d	+Noun	+Masc	+Pl	+NomAccGen
	↓	↓	↓	↓	↓	↓	↓	↓	↓
Surface:	G	r	ü	n	d	0	0	0	e

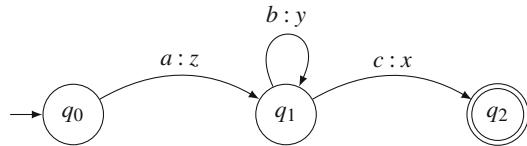
**Fig. 12.5** Alignments with features

morphs. For example, the Xerox parser (Beesley and Karttunen 2003) output for *disentangled*, *happier*, and *Gründe* is:

```
disentangle+Verb+PastBoth+123SP
happy+Adj+Comp
Grund+Noun+Masc+P1+NomAccGen
```

where the feature +Verb denotes a verb, +PastBoth, either past tense or past participle, and +123SP any person, singular or plural; +Adj denotes an adjective and +Comp, a comparative; +Noun denotes a noun, +Masc masculine, +P1, plural, and +NomAccGen either nominative, accusative, or genitive. (All these forms are ambiguous, and the Xerox parser shows more than one interpretation per form.)

Given these new lexical forms, the parser has to align the feature symbols with letters or null symbols. The principles do not change, however (Fig. 12.5).

**Fig. 12.6** A transducer

### 12.7.3 Finite-State Transducers

The two-level model is commonly implemented using finite-state transducers (FST). Transducers are automata that accept, translate, or generate pairs of strings. The arcs are labeled with two symbols: the first symbol is the input and the second is the output. The input symbol is transduced into the output symbol as a transition occurs on the arc. For instance, the transducer in Fig. 12.6 accepts or generates the string *abbbc* and translates into *zyyyx*.

Finite-state transducers have a formal definition, which is similar to that of finite-state automata. A FST consists of five components  $(Q, \Sigma, q_0, F, \delta)$ , where:

1.  $Q$  is a finite set of states.
2.  $\Sigma$  is a finite set of symbol or character pairs  $i : o$ , where  $i$  is a symbol of the input alphabet and  $o$  of the output alphabet. As we saw, both alphabets may include epsilon transitions.
3.  $q_0$  is the start state,  $q_0 \in Q$ .
4.  $F$  is the set of final states,  $F \subseteq Q$ .
5.  $\delta$  is the transition function  $Q \times \Sigma \rightarrow Q$ , where  $\delta(q, i, o)$  returns the state where the automaton moves when it is in state  $q$  and consumes the input symbol pair  $i : o$ .

The quintuple, which defines the automaton in Fig. 12.6 is  $Q = \{q_0, q_1, q_2\}$ ,  $\Sigma = \{a : z, b : y, c : x\}$ ,  $\delta = \{\delta(q_0, a : z) = q_1, \delta(q_1, b : y) = q_1, \delta(q_1, c : x) = q_2\}$ , and  $F = \{q_2\}$ .

### 12.7.4 Conjugating a French Verb

Morphological FSTs encode the lexicon and express all the legal transitions. Arcs are labeled with pairs of symbols representing letters of the surface form—the word—and the lexical form—the set of morphs.

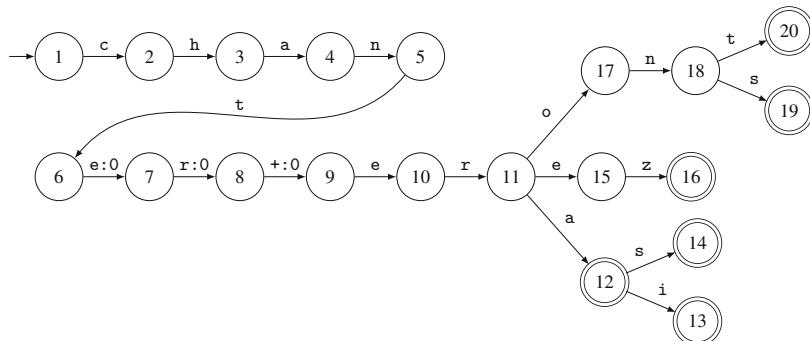
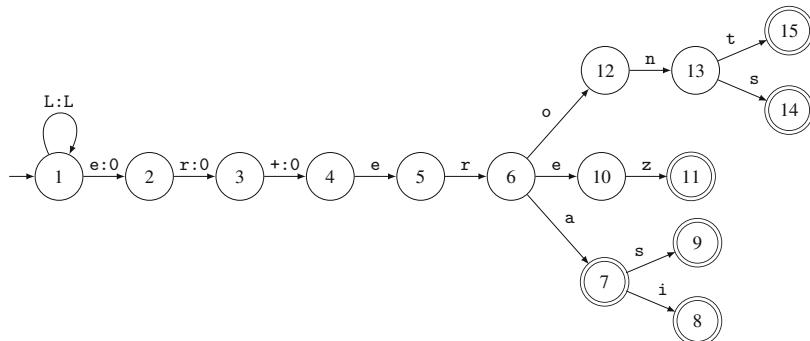
Table 12.27 shows the future tense of regular French verb *chanter* ‘sing’, where suffixes are specific to each person and number, but are shared by all the verbs of the so-called first group. The first group accounts for the large majority of French verbs. Table 12.28 shows the aligned forms and Fig. 12.7 the corresponding transducer. The arcs are annotated by the input/output pairs, where the left symbol corresponds to the lexical form and the right one to the surface form. When the lexical and surface characters are equal, as in *c:c*, we just use a single symbol in the arc.

**Table 12.27** Future tense of French verb *chanter*

Number\Person	First	Second	Third
Singular	<i>chanterai</i>	<i>chanteras</i>	<i>chantera</i>
Plural	<i>chanterons</i>	<i>chantereze</i>	<i>chanteront</i>

**Table 12.28** Aligned lexical and surface forms

Number\Pers.	First	Second	Third
Singular	<i>chanter+erai</i>	<i>chanter+eras</i>	<i>chanter+era</i>
	<i>chant000erai</i>	<i>chant000eras</i>	<i>chant000era</i>
Plural	<i>chanter+erons</i>	<i>chanter+erez</i>	<i>chanter+eront</i>
	<i>chant000erons</i>	<i>chant000erez</i>	<i>chant000eront</i>

**Fig. 12.7** A finite-state transducer describing the future tense of *chanter***Fig. 12.8** A finite-state transducer describing the future tense of French verbs of the first group

This transducer can be generalized to any regular French verb of the first group by removing the stem part and inserting a self-looping transition on the first state (Fig. 12.8).

The transducer in Fig. 12.8 also parses and generates forms that do not exist. For instance, we can forge an imaginary French verb *\*palimoter* that still can be conjugated by the transducer. Conversely, the transducer will successfully parse the

**Table 12.29** Future tense of Italian verb *cantare* and Spanish and Portuguese verbs *cantar*, ‘sing’

Language	Number\Person	First	Second	Third
Italian	Singular	<i>canterò</i>	<i>canterai</i>	<i>canterà</i>
	Plural	<i>canteremo</i>	<i>canterete</i>	<i>canteranno</i>
Spanish	Singular	<i>cantaré</i>	<i>cantarás</i>	<i>cantará</i>
	Plural	<i>cantaremos</i>	<i>cantaréis</i>	<i>cantarán</i>
Portuguese	Singular	<i>cantarei</i>	<i>cantarás</i>	<i>cantará</i>
	Plural	<i>cantaremos</i>	<i>cantareis</i>	<i>cantarão</i>

improbable *\*palimoterons*. This process is called overgeneration (both in parsing and generation).

Overgeneration is not that harmful, provided that inputs are well formed. However, it can lead to some wrong parses. Consider English and German comparatives that are formed with *-er* suffix. Raw implementation of a comparative transducer would rightly parse *greater* as *great+er* but could also parse *better* or *reader*. Overgeneration is reduced by a lexical lookup, where the parse result is searched in a dictionary. This eliminates nonexistent words. It can also be limited by a set of constraints on affixes restricting the part of speech of the word to which they can be appended—here adjectives.

### 12.7.5 Application to Romance Languages

The transducer we created for the conjugation of French verbs can be easily transposed to other Romance languages such as Italian, Spanish, or Portuguese, as shown in Table 12.29.

### 12.7.6 Ambiguity

In the transducer for future tense, there is no ambiguity. That is, a surface form has only one lexical form with a unique final state. This is not the case with the present tense (Table 12.30), and

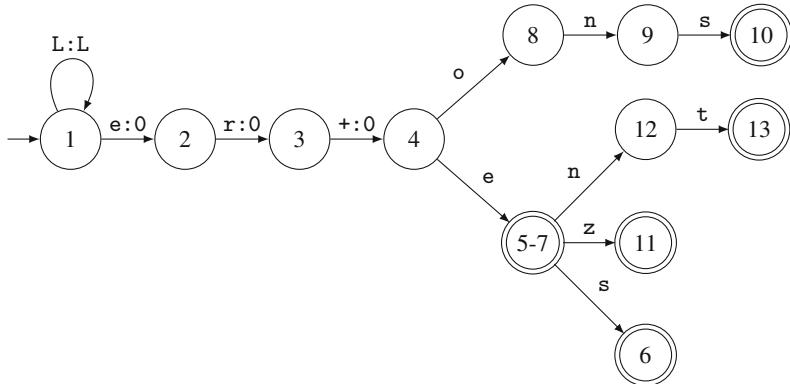
- (je) *chante* ‘I sing’
- (il) *chante* ‘he sings’

have the same surface form but correspond, respectively, to the first- and third-person singular.

This corresponds to the transducer in Fig. 12.9, where final states 5 and 7 are the same.

**Table 12.30** Present tense of French verb *chanter*

Number\Person	First	Second	Third
Singular	<i>chanте</i>	<i>chantes</i>	<i>chantе</i>
Plural	<i>chantons</i>	<i>chantez</i>	<i>chantent</i>



**Fig. 12.9** A finite-state transducer encoding the present tense of verbs of the first group

### 12.7.7 Operations on Finite-State Transducers

Finite-state transducers have mathematical properties similar to those of finite-state automata. In addition, they can be inverted and composed:

- Let  $T$  be a transducer. The inversion  $T^{-1}$  reverses the input and output symbols of the transition function. The transition function of the transducer in Fig. 12.6 is then  $\delta = \{\delta(q_0, z : a) = q_1, \delta(q_1, y : b) = q_1, \delta(q_1, x : c) = q_2\}$ .
- Let  $T_1$  and  $T_2$  be two transducers. The composition  $T_1 \circ T_2$  is a transducer, where the output of  $T_1$  acts as the input of  $T_2$ .

Both the inversion and composition operations result in new transducers. This is obvious for the inversion. The proof is slightly more complex for the composition. Let  $T_1 = (\Sigma, Q_1, q_1, F_1, \delta_1)$  and  $T_2 = (\Sigma, Q_2, q_2, F_2, \delta_2)$  be two transducers. The composition  $T_3 = T_1 \circ T_2$  is defined by  $(\Sigma, Q_1 \times Q_2, \langle q_1, q_2 \rangle, F_1 \times F_2, \delta_3)$ . The transition function  $\delta_3$  is built using the transition functions  $\delta_1$  and  $\delta_2$ , and generating all the pairs where they interact (Kaplan and Kay 1994):

$$\delta_3(\langle s_1, s_2 \rangle, i, o) = \{ \langle t_1, t_2 \rangle | \exists c \in \Sigma \cup \epsilon, t_1 \in \delta_1(s_1, i, c) \wedge t_2 \in \delta_2(s_2, c, o) \}.$$

The inversion property enables transducers to operate in generating or parsing mode. They accept both surface and lexical strings. Each symbol of the first string is mapped to the symbol of the second string. So you can walk through the automaton and retrieve the lexical form from the surface form, or conversely.

Composition enables us to break down morphological phenomena. It is sometimes easier to formulate a solution than using intermediate forms between the surface and lexical forms. The correspondence between the word form and the sequence of morphemes is not direct but is obtained as a cascade of transductions. Composition enables us to compact the cascade and to replace the transducers involved in it by a single one (Karttunen et al. 1992).

## 12.8 Further Reading

Dionysius Thrax fixed the parts of speech for Greek in the second century BCE. They have not changed since and his grammar is still interesting to read, see Lallot (1998). A short and readable introduction in French to the history of parts of speech is Ducrot and Schaeffer (1995).

Accounts on finite-state morphology can be found in Sproat (1992) and Ritchie et al. (1992). Roche and Schabes (1997) is a useful book that describes fundamental algorithms and applications of finite-state machines in language processing, especially for French. Kornai (1999) covers other aspects and languages. Kiraz (2001) on the morphology of Semitic languages: Syriac, Arabic, and Hebrew. Beesley and Karttunen (2003) is an extensive description of the two-level model in relation with the historical Xerox tools.

General-purpose finite-state transducers toolkits are available online. They include the FSA utilities (van Noord and Gerdemann 2001), the FSM library (Mohri et al. 1998) and its follower, OpenFst.<sup>2</sup>

Although the lemmatizers in the chapter used transducers and rules, it is possible to formulate lemmatization with classifiers that we can train on annotated corpora. See Chrupała (2006) for an interesting account on these techniques and Björkelund et al. (2010) for an implementation.

---

<sup>2</sup> <https://www.openfst.org/>.

# Chapter 13

## Subword Segmentation



In Chap. 12, *Words, Parts of Speech, and Morphology*, we used rules to split the words into morphemes. Figures in Table 12.24 show that this reduces considerably the size of the lexicon. Nonetheless, writing a morphological parser involves a linguistic knowledge, sometimes difficult to formulate and implement. In this chapter, we will deal with automatic techniques to extract subwords from a corpus, often matching morphemes, and enabling us to set a size limit to the lexicon.

We will first examine an algorithm that identifies morphemes using elementary statistics on the character distribution without any knowledge of linguistic rules. Then, we will study three other automatic methods, also based on statistics, to derive subwords, namely byte-pair encoding (Gage, 1994; Sennrich et al., 2016), WordPiece (Schuster & Nakajima, 2012), and Unigram/SentencePiece (Kudo, 2018; Kudo & Richardson, 2018).

Schuster and Nakajima (2012) applied their lexical model in voice search for Japanese and Korean. Their vocabulary consisted of 200,000 subwords shared between the two languages. Sennrich et al. (2016) showed that BPE could improve automatic translation while Wu et al. (2016) is another example of subword tokenization in translation.

### 13.1 Deriving Morphemes Automatically

Déjean (1998) proposed an algorithm to discover the morphemes of a language. As idea, he observed that morphemes correspond to fixed sequences of characters, where the letter preceding or following a morpheme, and thus marking the boundary, has a more random distribution. The input is a raw corpus and the algorithm consists of two steps: It selects a first set of morphemes from frequent word prefixes and word suffixes. It then complement this set with morphemes behaving like the frequent ones.

**Table 13.1** The most frequent morphemes

English	-e -s -ed -ing -al -ation -ly -ic -ent
French	-s -e -es -ent -er -ds -re -ation -ique
German	-en -e -te -ten -er -es -lich -el
Turkish	-m -in -lar -ler -dan -den -inl -ml
Swahili	-wa -ia -u -eni -o -isha -ana -we wa- m- ku- ali- ni- aka- ki- vi-

1. For the first step, let us begin with the word prefixes. The algorithm generates all the word prefixes starting with one character and, for each prefix, computes the distribution of the characters following it. If the number of characters in this distribution is greater than half the alphabet size and the most frequent next character represents less than 50% of the distribution, then the prefix is a morpheme. We apply the same algorithm to the word suffixes by reversing the words.
2. Next, we complement the list of morphemes by splitting each word into a prefix and a suffix. For a given prefix, if more than half of the suffixes belong to the morphemes discovered in step 1, we consider the rest of the suffixes as morphemes too. We also apply this rule to a given suffix to find the prefixed morphemes. Both steps have a cut off frequency.

Table 13.1 shows the most frequent morphemes Déjean (1998) found in a few languages.

Using these lists, we segment a word with the longest match algorithm. We implement it with a regular expression consisting of a disjunction of morphemes. To extract the suffixes, we need to reverse the corresponding strings. In the following code, we build a list of the English morphemes in Table 13.1 and we reverse them:

```
suffix_morphemes = ['-e', '-s', '-ed', '-ing',
                     '-al', '-ation', '-ly', '-ic', '-ent']
rev_morphemes = ['^' + suffix[1:][::-1] for
                  suffix in suffix_morphemes]
```

We sort the morphemes by decreasing length and we create the regex:

```
s_patterns = sorted(rev_morphemes,
                      key=lambda x: (-len(x), x))
s_regex = '|'.join(s_patterns)
```

We finally segment a word with the function:

```
import regex as re

re.findall(s_regex + '|\\p{L}+', word[::-1])
```

as with the word *celebration*:

```
list(map(lambda x: x[::-1], re.findall(s_regex +
                                         '|\\p{L}+', 'celebration'[::-1])[::-1]))
```

[‘celebr’, ‘ation’]

## 13.2 Byte-Pair Encoding

While Déjean (1998) designed the previous method with natural language processing in mind, Gage (1994) created byte-pair encoding (BPE) as a compression algorithm. His program reads a data sequence in the form of bytes and replaces the most frequent adjacent pair of bytes with a single byte not in the original data. This process repeats recursively and stores each pair it found with its replacement in a table. We can restore the original data from the table.

### 13.2.1 Outline of the Algorithm

Sennrich et al. (2016) adapted the original BPE algorithm to build automatically a lexicon of subwords from a corpus. These subwords consist of a single character, a sequence of characters, possibly a whole word, and the size of the lexicon is fixed in advance. The main steps of the algorithm are:

1. Split the corpus into individual characters. These characters will be the initial subwords and will make up the start vocabulary;
2. Then:
  - (a) Extract the most frequent adjacent pair from the corpus;
  - (b) Merge the pair and add the corresponding subword to the vocabulary;
  - (c) Replace all the occurrences of the pair in the corpus with the new subword;
  - (d) Repeat this process until we have reached the desired vocabulary size.

### 13.2.2 Pretokenization

BPE normally does not cross the whitespaces. This means that we can speed up the learning process with a pretokenization of the corpus and a word count.

The simplest pretokenization uses the whitespaces as delimiters, where we match the words, the numbers, and the rest, excluding the whitespaces, with this regular expression as in Sect. 9.2.1:

```
pattern = r'\p{L}+|\p{N}+|[\^s\p{L}\p{N}]+,'
```

To apply it to a corpus, we first read it as a string and store it in the `text` variable. We then pretokenize it with the following code, where we keep the word positions, as in Sect. 9.5.2:

```
import regex as re

words = [(match.group(), (match.start(), match.end()))
         for match in re.finditer(pattern, text)]
```

Applying this code to a corpus consisting of the *Iliad* and the *Odyssey*, we obtain the words:

```
[('BOOK', (0, 4)), ('I', (5, 6)), ('The', (10, 13)),
 ('quarrel', (14, 21)), ('between', (22, 29)),
 ('Agamemnon', (30, 39)), ('and', (40, 43)),
 ('Achilles', (44, 52)), ...]
```

The word positions would enable us to restore the original text. Nonetheless, in the rest of this section, we set them aside to simplify the code and we define a `pretokenize()` function with `.findall()` instead:

```
def pretokenize(pattern, text):
    return re.findall(pattern, text)
```

This pretokenization is quite rudimentary as it would create a single token for sequence of punctuations. We could improve it and, for instance, extend the regular expression to create a token for each punctuation symbol:

```
r'\p{L}+|\p{N}+|\p{P}|[\^s\p{L}\p{N}\p{P}]+'
```

In the rest of the implementation, we will use a class to encapsulate the BPE functions. So far, the class only contains the pretokenization and we define the pretokenization pattern when we create a BPE object:

```
class BPE():
    def __init__(self):
        self.pattern = r'\p{L}+|\p{N}+|[\^s\p{L}\p{N}]+'

    def pretokenize(self, text):
        return re.findall(self.pattern, text)
```

We will add the other functions we need along with their description in the next sections.

### 13.2.3 The Initial Vocabulary

The initial vocabulary consists of the individual characters from which we build iteratively the subwords. For this, we create a dictionary, `words_bpe`, where we associate the words with their subwords, starting with these characters. As key, we use the word string and as value, a dictionary with the word frequency, `freq`, and the list of subwords in construction, `swords`.

We count the words from the pretokenized text with the `Counter` class:

```
>>> from collections import Counter

>>> bpe = BPE()
>>> words = bpe.pretokenize(text)
>>> word_cnts = Counter(words)
>>> word_cnts.most_common(5)
```

yielding:

```
[(' ', 19518), ('the', 15311), ('and', 11521),
 ('of', 8677), ('.', 6870)]
```

Using `word_cnts`, we create a new function to initialize the subwords that we add to the class. We also extracts the set of characters from the corpus that we assign to `self.vocab`:

```
def _bpe_init(self, text):
    words = self pretokenize(text)
    word_cnts = Counter(words)
    self.words_bpe = {
        word: {'freq': freq,
               'swords': list(word)}
        for word, freq in word_cnts.items()}
    self.vocab = list(
        set([char for word in self.words_bpe
              for char in self.words_bpe[word]['swords']])))
```

Running:

```
>>> bpe = BPE()
>>> bpe._bpe_init(text)
```

we have the dictionary entry:

```
>>> bpe.words_bpe['her']
{'freq': 1147, 'swords': ['h', 'e', 'r']}
```

Once we have passed this initial step, we can implement the vocabulary construction. This is a loop, where at a given iteration, we update the `swords` value by merging the most frequent pair of subwords.

### 13.2.4 Counting the Bigrams

We count the adjacent pair of symbols, either characters or subwords, with this function:

```
def _count_bigrams(self):
    self.pair_cnts = Counter()
    for word_dict in self.words_bpe.values():
        swords = tuple(word_dict['swords'])
        freq = word_dict['freq']
        for i in range(len(swords) - 1):
            self.pair_cnts[swords[i:i + 2]] += freq
```

At the first iteration, we have:

```
>>> bpe = BPE()
>>> bpe._bpe_init(text)
>>> bpe._count_bigrams()
>>> max(bpe.pair_cnts, key=bpe.pair_cnts.get)
('h', 'e')
```

### 13.2.5 Merging Pairs

We merge the pair ('h', 'e') in all the subwords of `words_bpe` so that

```
[‘h’, ‘e’, ‘r’]
```

becomes:

```
[‘he’, ‘r’]
```

We apply this operation with the function below, where the pair and the subwords are lists. We traverse the subwords and we extend a new list with its items or the pair if it matches two adjacent items:

```
def _merge_pair(self, pair, swords):
    pair_str = ''.join(pair)
    i = 0
    temp = []
    while i < len(swords) - 1:
        if pair == swords[i:i + 2]:
            temp += [pair_str]
            i += 2
        else:
            temp += [swords[i]]
            i += 1
    if i == len(swords) - 1:
        temp += [swords[i]]
    swords = temp
    return swords
```

When applying this operation to `they`, we have:

```
>>> bpe._merge_pair(['h', 'e'], ['t', 'h', 'e', 'y'])
['t', 'he', 'y']
```

### 13.2.6 Constructing the Merge Rules

We can now build the list of merges with a function that gets the most frequent adjacent pair of subwords, merges it in all the `words_bpe` dictionaries, and repeats the process until we have reached the predetermined vocabulary size. Let us call this function `fit()`. When it terminates, we add the pairs to the initial set of characters with `_build_vocab()`:

```
def fit(self, text):
    self._bpe_init(text)

    self.merge_ops = []
    for _ in range(self.merge_cnt):
        self._count_bigrams()
        self.best_pair = max(self.pair_cnts,
                             key=self.pair_cnts.get)
```

```

        merge_op = list(self.best_pair)
        self.merge_ops.append(merge_op)
        for word_dict in self.words_bpe.values():
            word_dict['swords'] = self._merge_pair(
                merge_op,
                word_dict['swords'])
    self._build_vocab()

def _build_vocab(self):
    swords = list(map(lambda x: ''.join(x), self.merge_ops))
    self.vocab += swords

```

Before we can run this function, we need to modify the `__init__()` function to give the number of merges:

```

def __init__(self, merge_cnt=200):
    self.pattern = r'\p{L}+|\p{N}+|[\^s\p{L}\p{N}]+' 
    self.merge_cnt = merge_cnt

```

We run our program with:

```

>>> bpe = BPE()
>>> bpe.fit(text)

```

and our corpus of Homer works, the five first merges in `bpe.merge_ops[:5]` are:

```

[['h', 'e'], ['t', 'he'], ['a', 'n'], ['i', 'n'], ['o', 'u']]

```

### 13.2.7 Encoding

Once we have derived a list of merge operations, we can apply these operations to the characters of a word in the same order we created them:

```

def encode(self, word):
    swords = list(word)
    for op in self.merge_ops:
        swords = self._merge_pair(op, swords)
    return swords

```

For *therefore*, this yields:

```

>>> bpe = BPE()
>>> bpe.fit(text)
>>> bpe.encode('therefore')
['there', 'fore']

```

In `fit()`, we have defined a vocabulary consisting of the characters in the training corpus and the subwords. With further processing, we could map the characters outside this set to 'UNK'. Otherwise the initial vocabulary consists of all the Unicode characters.

### 13.2.8 Tokenizing

To tokenize a text, we first apply a pretokenization. We then encode all its words. To speed up the merges, we use a cache:

```
def tokenize(self, text):
    tokenized_text = []
    cache = {}
    words = self.pretokenize(text)
    for word in words:
        if word not in cache:
            cache[word] = self.encode(word)
        subwords = cache[word]
        tokenized_text += subwords
    return tokenized_text
```

We can now tokenize a whole text, for instance this quote from Virgil:<sup>1</sup>

Exiled from home am I; while, Tityrus, you  
Sit careless in the shade

```
bpe = BPE(pattern)
bpe.fit(text)
ecloges_str = """Sit careless in the shade"""

>>> bpe.tokenize(ecloges_str)
['S', 'it', 'c', 'are', 'le', 's', 's', 'in', 'the', 's',
 'had', 'e']
```

### 13.2.9 Visualizing the Whitespaces

In the tokenized string of the last paragraph, we have lost track of the visual word separators. This makes it difficult to read and can be a bit confusing. GPT2 (Radford et al., 2019) provides a solution to this with a pretokenization that creates initial tokens with a prefixed whitespace and then replaces the whitespaces with the  $\text{\texttt{G}}$  character.

This idea is easy to implement. We just need to modify the pretokenization pattern so that it matches one possible leading whitespace:

```
pattern = r' ?\p{L}+| ?\p{N}+| ?[^s\p{L}\p{N}]+'
```

and the pretokenization function to translate the leading whitespaces. We add a `leading_space` argument to select this kind of pretokenization:

---

<sup>1</sup> Nos patriam fugimus; tu, Tityre, lensus in umbra, Virgil, *Eclogues*, Rhoades, James, translator. London: Oxford University Press, 1921.

```

def __init__(self, merge_cnt=200, leading_space=False):
    self.merge_cnt = merge_cnt
    self.leading_space = leading_space
    if leading_space:
        self.pattern = r' ?\p{L}+| ?\p{N}+| ?[^\\s\\p{L}\\p{N}]+'
    else:
        self.pattern = r'\\p{L}+|\\p{N}+|[^\\s\\p{L}\\p{N}]+'

def pretokenize(self, text):
    words = re.findall(self.pattern, text)
    if self.leading_space:
        words = [''.join((self.G, word[1:])) for word in words]
        if word[0] == ' ' else word
    return words

```

We run the program again on the corpus with this pretokenization and we obtain new merge rules. We then apply these rules to tokenize Virgil’s verse yielding:

```
[‘S’, ‘it’, ‘Gc’, ‘a’, ‘re’, ‘l’, ‘ess’, ‘Gin’, ‘Gthe’,
 ‘Gsh’, ‘ad’, ‘e’]
```

where we see more easily the words.

### 13.2.10 Using Bytes

So far, we have used the Unicode characters as our initial vocabulary. As Unicode has about 150,000 characters, this can lead to a very large lexicon if we want to train BPE on any kind of text, in any language. That is why the GPT-2 tokenizer uses the original byte-level encoding of Gage (1994) and restricts the initial symbol set to the 256 possible bytes.

To make the bytes printable, the GPT-2 pretokenization replaces all the ASCII and Latin-1 control or nonprintable characters with a codepoint shifted by 256. The codepoint ranges of these characters are [0, 32] and [127, 160], plus the soft-hyphen, 173, see Tables 4.1 and 4.3. In total, there are 68 characters that we number from 0 to 67. We shift them with the function `chr(n + 256)`, where `n` is their rank, 0 to 67. For the whitespace in the previous section, the first of these shifted characters, we have:

```
>>> chr(ord(' ') + 256)
'G'
```

While this technique results in a smaller lexicon, it impairs the legibility of non-ASCII characters, and thus of languages outside English. For instance, the UTF-8 codes of letters like é or ä consist two bytes and are rendered as ‘Ã©’ and ‘Ã¤’. Indic or Chinese characters lead to three or four such bytes.

### 13.3 The WordPiece Tokenizer

WordPiece (Schuster & Nakajima, 2012; Wu et al., 2016) is a tokenizer similar to BPE: It builds incrementally a lexicon of subwords from a corpus by merging adjacent pairs. It then uses these subwords to tokenize text. There are two major differences however:

1. The criterion to merge a pair is the quality of the resulting language model; see Chap. 10, *Word Sequences*;
2. The tokenization uses a greedy longest match algorithm.

An exact comparison of language models for each pair would be exceptionally expensive. That is why Schuster and Nakajima (2012) applied a series of simplifications that, unfortunately, they did not describe in detail.

In the absence of any further information, we will conjecture that WordPiece uses a simple unigram language model. This will speed up the computation. The likelihood of a subword sequence of length  $N$ ,  $(x_1, x_2, \dots, x_N)$ , is then  $\prod_{i=1}^N P(x_i)$ ; using logarithms, this is equivalent to  $\sum_{i=1}^N \log P(x_i)$ . Given an adjacent pair of subwords  $(x, y)$  with  $C(xy)$  occurrences in the original sequence, the log-likelihood of the model is equivalent to:

$$\sum_{\substack{i=1 \\ x_i, x_{i+1} \neq x, y}}^N \log P(x_i) + C(xy)(\log P(x) + \log P(y)).$$

When we merge this pair, the log-likelihood becomes:

$$\sum_{\substack{i=1 \\ x_i, x_{i+1} \neq x, y}}^N \log P(x_i) + C(xy) \log P(xy).$$

The difference between these two log-likelihoods is then:

$$C(xy) \cdot (\log P(xy) - \log P(x) - \log P(y)).$$

To build the lexicon, we proceed iteratively as with BPE and, at each step, we select the pair that improves the most the criterion above.

#### 13.3.1 Pretokenization and Initial Vocabulary

Similarly to BPE, WordPiece uses a whitespace pretokenization, corresponding to the `str.split()` function, followed by a tokenization of the punctuation (Google, 2019). We carry this out with the regex:

```
pattern = r'\p{P}|\[^\\s\\p{P}]+'
```

We count the pretokenized words and we store them in a data structure identical to that of BPE. As with BPE, we keep track of the start of a word, this time with the ‘  ’ (U+2581) character.

The WordPiece code to pretokenize a text and count the words is very similar to that of BPE. We also extracts the set of characters from the corpus. This forms the initial vocabulary:

```
class WordPiece():
    def __init__(self, merge_cnt=200):
        self.pattern = r'\p{P}|[^\\s\p{P}]+'
        self.merge_cnt = merge_cnt

    def pretokenize(self, text):
        words = re.findall(self.pattern, text)
        words = list(map(lambda x: ' ' + x, words))
        return words

    def _wp_init(self, text):
        words = self.pretokenize(text)
        word_cnts = Counter(words)
        self.words_wp = {
            word: {'freq': freq, 'words': list(word)}
            for word, freq in word_cnts.items()}
        self.vocab = list(
            set([char for word in self.words_wp
                  for char in self.words_wp[word]['words']]))


```

Running it on Homer’s works:

```
>>> wp = WordPiece()
>>> wp._wp_init(text)
>>> wp.words_wp
```

this yields:

```
{{'__BOOK': {'freq': 24, 'words': [' ', 'B', 'O', 'O', 'K']}, 
'__I': {'freq': 3202, 'words': [' ', 'I']}, 
'__The': {'freq': 548, 'words': [' ', 'T', 'h', 'e']}, 
'__quarrel': {'freq': 28, 'words': [' ', 'q', 'u', 'a', 'r', 'r', 'e', 'l']}, 
...}
```

### 13.3.2 Computing the Gains

Given a pair of adjacent symbols, we can now calculate the model likelihood before they are merged and after it. We do this for all the pairs:

```
def _calc_pair_gains(self):
    sword_cnts = Counter()
    self.pair_gains = Counter()
    for word_dict in self.words_wp.values():
        subwords = tuple(word_dict['words'])
```

```

freq = word_dict['freq']
for i in range(len(subwords) - 1):
    sword_ctns[subwords[i]] += freq
    self.pair_gains[subwords[i:i + 2]] += freq
    sword_ctns[subwords[len(subwords) - 1]] += freq
pair_cnt = sum(self.pair_gains.values())
sword_cnt = sum(sword_ctns.values())
for pair in self.pair_gains:
    self.pair_gains[pair] *= (
        log(self.pair_gains[pair]/pair_cnt)
        - log(sword_ctns[pair[0]]/sword_cnt)
        - log(sword_ctns[pair[1]]/sword_cnt))

```

We rank the pairs by decreasing order of improvement. For Homer's *Iliad* and *Odyssey*, this gives:

```

>>> wp = WordPiece()
>>> wp._wp_init(text)
>>> wp._calc_pair_gains()
>>> sorted(wp.pair_gains, key=wp.pair_gains.get, reverse=True)
[('t', 'h'), ('h', 'e'), ('a', 'n'), ('_', 't') ...]

```

### 13.3.3 Constructing the Subwords

We select the best pair from this list and merge all its occurrences in the subwords of `self.words_wp`. We use the same function as with BPE:

```
_merge_pair(self, pair, swords)
```

We repeat this operation in a loop, where, at each iteration, we select the pair that improves most the language model and we merge it in all the words. We stop the loop when we have reached the predefined number of subwords, here 200:

```

def fit(self, text):
    self._wp_init(text)

    self.merge_ops = []
    for _ in range(self.merge_cnt):
        self._calc_pair_gains()
        self.best_pair = max(self.pair_gains,
                             key=self.pair_gains.get)
        merge_op = list(self.best_pair)
        self.merge_ops.append(merge_op)
        for word_dict in self.words_wp.values():
            word_dict['swords'] = self._merge_pair(merge_op,
                                                    word_dict['swords'])

    self._build_vocab()

def _build_vocab(self):
    swords = list(map(lambda x: ''.join(x), self.merge_ops))
    self.vocab += swords

```

Once we have extracted the subwords, we add them to the initial symbols to build the final vocabulary: `self.vocab`.

For Homer's works, the first merges are:

```
>>> wp = WordPiece()
>>> wp.fit(text)
>>> wp.merge_ops
[['t', 'h'], ['th', 'e'], ['a', 'n'], ['an', 'd'], ...]
```

### 13.3.4 Encoding

The tokenization uses a whitespace pretokenization (Google, 2019) and then a greedy longest match that we can implement with a regular expression. To create the regex, we sort the vocabulary by length of the subwords and we create a disjunction of all the strings. We escape them to match literally the metacharacters:

```
def _create_regex(self):
    self.vocab.sort(key=lambda x: -len(x))
    # We escape metachars as for '.'
    self.vocab = [re.escape(word) for word in self.vocab]
    self.sword_regex = '|'.join(self.vocab)
```

We call this `_create_regex()` function at the end of `_build_vocabulary()`:

```
def _build_vocab(self):
    swords = list(map(lambda x: ''.join(x), self.merge_ops))
    self.vocab += swords
    self._create_regex()
```

Running the fitting procedure again, we have:

```
>>> wp = WordPiece()
>>> wp.fit(text)
>>> wp.sword_regex
'__Trojans|__should|__their|__which|__would|__about|__shall| ...
```

We find all the subwords with `re.findall()` as with

```
>>> re.findall(wp.sword_regex, '__Therefore')
['__The', 're', 'fore']
```

The match can fail to segment a word if it contains characters not in the vocabulary. For example, in the corpus we used, there is no é and the segmentation of `__touché` results in:

```
>>> re.findall(wp.sword_regex, '__touché')
['__to', 'u', 'ch']
```

where é is missing in the end.

To handle these cases, we add an unknown symbol to the class and we replace the whole word with it:

```
self.unk_word = '[UNK]'
```

We can now write a complete encoding function that checks that the segmentation does not lose any character:

```
def encode(self, word):
    subwords = re.findall(self.sword_regex, word)
    if ''.join(subwords) != word:
        # some subwords are not in the vocabulary
        subwords = [self.unk_word]
    return subwords
```

We have

```
>>> wp.encode('__Therefore')
['__The', 're', 'fore']
>>> wp.encode('__touché')
['__[UNK]']
```

### 13.3.5 Tokenization

Finally, we write a complete tokenization function. It uses a cache to avoid recomputations and is identical to that of BPE:

```
def tokenize(self, text):
    tokenized_text = []
    cache = {}
    words = self.pretokenize(text)
    for word in words:
        if word not in cache:
            cache[word] = self.encode(word)
        subwords = cache[word]
        tokenized_text += subwords
    return tokenized_text
```

The tokenization of Virgil's verse produces:

```
>>> wp = WordPiece()
>>> wp.fit(text)
>>> swords = wp.tokenize('Sit careless in the shade!')
>>> swords
['__S', 'it', '__c', 'ar', 'e', 'le', 's', 's', '__in',
 '__the', '__sh', 'ad', 'e', '__', '!']
```

### 13.3.6 BERT's WordPiece

The most popular implementation of WordPiece is probably that of in BERT. In this variant, instead of using the `__` prefix to mark the start of a word, the tokenizer visualizes the continuation segments with the `##` sequence. We convert our notation to BERT's with this function:

```
def bert_wp(swords):
    i = 0
    while i < len(swords) - 1:
        if swords[i] == '_':
            swords = swords[:i] + \
                [''.join([swords[i], swords[i + 1]])] + swords[i + 2:]
        i += 1
    return [sword[1:] if sword[0] == '_' else '##' + sword
            for sword in swords]
```

and we have:

```
>>> bert_wp(['S', '##it', 'c', '##ar', '##e', '##le', '##s', '##s',
             'in', 'the', 'sh', '##ad', '##e', '!'])
```

## 13.4 Unigram Tokenizer

It is often the case that we can segment a word in multiple ways. For example, given the vocabulary  $V = \{t, h, e, th, he, the\}$ , the word *the* has four possible tokenizations:

```
['t', 'h', 'e']
['th', 'e']
['t', 'he']
['the']
```

BPE and WordPiece always produce the same results as they use a deterministic algorithm, either by applying a list of merge operations or by finding the longest matches. In this section, we will describe an algorithm that maximizes a unigram language model (Kudo, 2018). This means that, for a given word, out of all the possible segmentations, the tokenizer keeps the one that has the maximal likelihood.

As with BPE, we start with a pretokenization into words. Then, given a vocabulary  $V$  of subwords and a word  $w$ , where

$$w = \text{concat}(sw_1, sw_2, \dots, sw_n) \text{ and } \forall i \in 1..n, sw_i \in V,$$

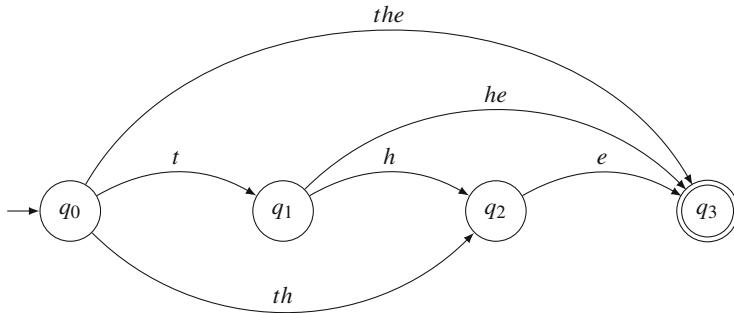
the unigram tokenization of  $w$  is the subword sequence  $sw_1, sw_2, \dots, sw_n$  that maximizes

$$\prod_{i=1}^n P(sw_i)$$

over all the possible subwords.

Using our previous example with  $V = \{t, h, e, th, he, the\}$  and the word *the*, the tokenizer evaluates four products:

1.  $P(t) \times P(h) \times P(e),$
2.  $P(th) \times P(e),$



**Fig. 13.1** Possible segmentations of *the*

3.  $P(t) \times P(he)$ ,
4.  $P(the)$ ,

and selects the highest probability. These possible tokenizations correspond to the transitions shown in Fig. 13.1.

To prevent an arithmetic underflow, in our implementation, as in Sect. 10.3, we will replace the product with a negative sum of logarithms, the negative log-likelihood (NLL):

$$-\sum_{i=1}^n \log P(sw_i)$$

that we will minimize.

### 13.4.1 Initial Class

For this description, we can create our initial unigram class with a pretokenization that will use this pattern:

```
words = re.findall(r'\p{P}|[^\\s\\p{P}]+' , text)
```

and where we append the ‘\_’ prefix to the words as with WordPiece.

```
class Unigram():
    def __init__(self, uni_probs):
        self.uni_probs = uni_probs
        self.pattern = r'\p{P}|[^\\s\\p{P}]+'

    def pretokenize(self, text):
        words = re.findall(self.pattern, text)
        words = list(map(lambda x: '_' + x, words))
        return words
```

Nonetheless, so far, we cannot create a `Unigram` object as we do not have an estimate of the unigram probabilities. This will be the topic of the next section.

### 13.4.2 Estimating the Probabilities

To compute the likelihood of a segmentation, we need an estimate of the probabilities of the subwords, for instance  $P(th)$ . As a word has multiple possible segmentations, we cannot directly estimate these probabilities. We will proceed iteratively:

1. We first calculate an estimation of the subword distribution with an initial BPE segmentation;
2. We then use these estimates to find a segmentation of the words that maximizes the unigram probabilities;
3. We use this segmentation to compute a new estimation of the subword distribution;
4. We repeat this procedure from step 2. until the subword probability distribution converges.

This procedure is called an **expectation-maximization**. In practice, we repeat it only a few times.

We start with the first step, where we tokenize the text with BPE. As the unigram language model marks the start of a word with '`_`', we modify the pretokenization of the BPE class in Sect. 13.2:

```
class BPE():
    def __init__(self, merge_cnt=200):
        self.merge_cnt = merge_cnt
        self.pattern = r'\p{P}|[^\\s\p{P}]+'

    def pretokenize(self, text):
        words = re.findall(self.pattern, text)
        words = list(map(lambda x: '_' + x, words))
        return words
```

the rest of the class is the same and we can fit the BPE model:

```
bpe = BPE()
bpe.fit(text)
tokens = bpe.tokenize(text)
```

We then count the words and we create a dictionary with the negative logarithms of the relative frequencies. We use this function:

```
def calc_nll(tokens):
    token cnts = Counter(tokens)
    total_cnt = token cnts.total()
    uni_probs = {token: -log(cnt/total_cnt) for
                 token, cnt in token cnts.items()}
    return uni_probs
```

that we apply to the corpus:

```
>>> uni_probs_bpe = calc_nll(tokens)
>>> uni_probs_bpe['her']
5.885
```

These values will suffice to run a first iteration of the unigram segmentation. We will describe the next steps of expectation-maximization in Sect. 13.4.4 when we have tokenized a text with the initial frequencies.

### 13.4.3 Finding the Best Segmentation

Now that we have devised a way to build an initial vocabulary, let us dive into the word tokenization. We will examine two techniques: one with a brute-force search of all the possible segmentations and a more frugal method with the Viterbi algorithm.

#### Brute-Force Search

Given a word, a brute-force algorithm will generate all the possible sequences of substrings. For this, we model the splits with Booleans that we insert between the characters. In Fig. 13.1, the intercharacters would correspond to the nodes between the start and the end nodes, i.e.  $q_1$  and  $q_2$ . We then split the string between two characters if the Boolean value is 1.

By varying all the values, we enumerate all the possible splits. Table 13.2 shows the subwords of *the* with the four possible values of  $(q_1, q_2)$ .

Let us now write the `split_word()` function that splits a `string` according to `splitpoints` given as a string of binary digits. We just go through the split points and create a new list if the value is 1:

```
@staticmethod
def split_word(string, splitpoints):
    subwords = []
    prev_sp = 0
    for i, sp in enumerate(splitpoints, start=1):
        if sp == '1':
            subword = string[prev_sp:i]
            prev_sp = i
            subwords.append(subword)
    subword = string[prev_sp:]
    subwords.append(subword)
    return subwords
```

**Table 13.2** The possible splits of *the*, where  $q_1$  and  $q_2$  are nodes in the automaton in Fig. 13.1. A 1 creates a split and 0, no split

$q_1$	$q_2$	Resulting splits
0	0	[‘the’]
0	1	[‘th’, ‘e’]
1	0	[‘t’, ‘he’]
1	1	[‘t’, ‘h’, ‘e’]

Let us incorporate this function as a static method in the `Unigram` class. For *there*, we model the split points as a string of four Booleans, for example '0110':

```
>>> Unigram.split_word('there', '0110')
['th', 'e', 're']
```

To produce the strings of binary digits, such as '0110', we will use Python's formatted string literals. These literals consist of an `f` prefix and a content enclosed in curly braces, `f'{number:formatting}'`. The content is a number with formatting specifications. The format specifier is a mini-language and has many options. Here we will use the literal: `f'{number:0{number_of_bits}b}'` telling to format `number` on the specified number of binary digits and padded with zeros.

For a word with  $n$  characters, we have  $2^{n-1}$  sequences of binary digits corresponding to the possible split points. This is a variant of the powerset, here applied to subsequences. To create the segmentations, we generate all the numbers between 0 and  $2^{n-1}$  encoded on  $n - 1$  bits and we call `Unigram.split_word()`:

```
word = 'there'
cnt_sp = len(word) - 1
candidates = []
for i in range(2**cnt_sp):
    splitpoints = f'{i:0{cnt_sp}b}'
    candidates += [Unigram.split_word(word, splitpoints)]
candidates
```

resulting in

```
[['there'],
 ['ther', 'e'],
 ['the', 're'],
 ['the', 'r', 'e'],
 ...
 ['t', 'h', 'e', 'r', 'e']]
```

Now that we have these sequences, we can compute the maximal likelihood of a word segmentation, or here the minimal sum of logarithms. Our brute-force tokenization generates all the segmentations, evaluates their likelihoods, and returns the most likely one. We limit the length of a word to 20 characters.

```
def encode(self, word):
    cnt_sp = len(word) - 1
    if cnt_sp > 20:
        return list(word)
    candidates = []
    for i in range(2**cnt_sp):
        splitpoints = f'{i:0{cnt_sp}b}'
        candidates += [Unigram.split_word(word, splitpoints)]
    return min(
        [(cand,
            sum(map(lambda x: self.uni_probs.get(x, 1000), cand))
            ) for cand in candidates],
        key=lambda x: x[1])
```

When a segment is not in the vocabulary, we assign it a very unlikely value: 1000.

We then apply the tokenization to a text with the method:

```
def tokenize(self, text):
    cache = {}
    nll_text = 0.0
    tokenized_text = []
    words = self pretokenize(text)
    for word in words:
        if word not in cache:
            cache[word] = self.encode(word)
        subwords, nll = cache[word]
        tokenized_text += subwords
        nll_text += nll
    return tokenized_text, nll_text
```

This brute-force search has an exponential complexity. This makes it prohibitive and unusable in most practical cases. In the next section, we will look at the Viterbi algorithm (Viterbi, 1967) that will provide us with a solution.

## Viterbi Algorithm

Our second algorithm will start with an empty list and incrementally add subwords. It will evaluate the likelihood of incomplete sequences and eliminate those that have no chance of maximizing the likelihood.

Consider *there*, for instance. After two characters, we can compare two segmentations: [‘\_t’, ‘h’] and [‘\_th’]. Trained on Homer’s works with BPE, the negative log likelihoods of \_t, h, and \_th are respectively 5.11, 5.25, and 5.57. At this point in the segmentation, there is no way  $5.11 + 5.25 = 10.36$  can be better than 5.57. We can then discard [‘\_t’, ‘h’]. We will store the intermediate result that, at index 2, is the best segmentation [‘\_th’] and reuse it when we proceed further in the word. This is the idea of the Viterbi algorithm (Viterbi, 1967). Using Viterbi’s words, we will call the intermediate result at index  $i$ , a survivor.

As data structure, given a word of  $n$  character, we will use two lists: one to store the surviving subwords at index  $i$ , `swords`, and one to store the minimal negative log likelihoods, `min_nlls`. We initialize the first list with the input words truncated from index  $i$  to its end: `word[:i]`, i.e. longer and longer prefixes, and the second with the negative log likelihoods of these subwords.

Following our example, we implement the algorithm with two loops. The first one goes through the characters of the word using index  $i$ ; the second loop with index  $j$  goes through the characters from 1 up to  $i$  and computes the likelihood of the `word[j:i]` subword. If the resulting segmentation is better, we update the NLL of the word at index  $i - 1$ . We use here a dynamic programming technique: Instead of recomputing the negative log likelihoods, we look them up in this list.

Eventually, when we have reached the end of the word, we extract the subwords. For this, we start from the end, where we have the last subword. We remove the

incomplete sequences leading to this last subword. We repeat this process until we have reached the beginning of the list.

```
def encode(self, word):
    n = len(word)
    swords = [word[:i] for i in range(1, n + 1)]
    min_nlls = [self.uni_probs.get(sword, 1000.0) for sword in swords]

    for i in range(2, n + 1):
        for j in range(1, i):
            sword = word[j:i]
            nll = self.uni_probs.get(sword, 1000.0) + min_nlls[j - 1]
            if min_nlls[i - 1] > nll:
                min_nlls[i - 1] = nll
                swords[i - 1] = sword

    # backtrace
    final_swords = [swords.pop()]
    while True:
        for i in range(len(final_swords[-1]) - 1):
            swords.pop()
        if swords:
            final_swords += [swords.pop()]
        else:
            break
    return final_swords[::-1], min_nlls[-1]
```

We replace the brute-force encode method with the new one:

```
>>> unigram = Unigram(uni_probs_bpe)
>>> unigram.encode('therefore')
(['ther', 'e'], 10.34257257624196)
>>> unigram.tokenize('Sit careless in the shade')
(['_S', 'it', '_c', 'a', 're', 'le', 's', 's', '_in',
 '_the', '_sh', 'ad', 'e'], 61.6840259320422)
```

### 13.4.4 Expectation-Maximization (EM)

In Sect. 13.4.2, we outlined how to estimate the probabilities. Now that we have ran a first segmentation with BPE estimates (`uni_probs_old`), we can recompute new estimates (`uni_probs_new`). This corresponds to the function below.

```
def em(text, uni_probs_old):
    cache = {}
    tokens = []
    unigram = Unigram(uni_probs_old)
    words = unigram.pretokenize(text)
    for word in words:
        if word not in cache:
            cache[word] = unigram.encode(word)[0]
        tokens += cache[word]
    uni_probs_new = calc_nll(tokens)
    return uni_probs_new
```

We can then reapply a segmentation and repeat this procedure a few times, here 5, or until the estimates have converged, i.e. do not change between two tokenizations.

```
uni_probs = dict(uni_probs_bpe)
for _ in range(5):
    uni_probs = em(words, uni_probs)
```

### 13.4.5 Creating the Vocabulary

Finally, so far, we do not know the optimal vocabulary of subwords for a unigram segmentation. In the previous sections, we have assumed that we could obtain it with BPE. This is nonetheless an approximation only. For a given vocabulary size, we would need to generate and test all the possible subword combinations to find it. In practice, this is impossible.

As a feasible implementation, Kudo (2018) proposed to start with a superset of the final vocabulary that we obtain with another method, here BPE. This superset,  $V$ , is the seed vocabulary and its size should be reasonably large to be sure it contains the optimal subset. Then, we remove the words from  $V$  by order of least significance for the quality of the language model until we reach the desired size. We always keep subwords of length 1 to avoid out-of-vocabulary words.

We implement the elimination incrementally: For all the subwords  $sw_i \in V$ , we compute the NLL value on a corpus with a vocabulary deprived from this subword,  $V \setminus \{sw_i\}$ :

$$\text{NLL}(\text{corpus}, V \setminus \{sw_i\}).$$

We measure the contribution of  $sw_i$  to the performance of the language model with this loss equation:

$$L_{\text{corpus}}(V \setminus \{sw_i\}, V) = \text{NLL}(\text{corpus}, V \setminus \{sw_i\}) - \text{NLL}(\text{corpus}, V),$$

i.e. how much we lose without it.

A high loss means that the language model is better with  $sw_i$  as the likelihood decreases (remember that we use the negative likelihood). A low or negative loss means that  $sw_i$  has only a small influence or even degrade the language model and we can discard it.

To build the final vocabulary, we sort the NLL values and we remove one or more subwords yielding the lowest values:

$$\arg \min_i \text{NLL}(\text{corpus}, V \setminus \{sw_i\}).$$

We apply the expectation-maximization algorithm to the new vocabulary to compute the probability estimates of the remaining subwords. We repeat these steps until we have reached the desired size.

## 13.5 The SentencePiece Tokenizer

SentencePiece (Kudo & Richardson, 2018) is a final subword tokenizer using either BPE or a unigram language model. The main difference with the previous algorithms is that there is no pretokenization. This is especially useful for languages like Chinese and Japanese, where there is no space between the words. SentencePiece considers the white spaces as other characters.

As preprocessing, SentencePiece replaces all the white spaces with ‘\_’ (U+2581). We can then train it on raw corpora. The decoding of tokenized text is very easy. Kudo and Richardson (2018) just replace the ‘\_’ characters with spaces with this statement:

```
detok = ''.join(tokens).replace(' ', '_')
```

SentencePiece has become one of the most popular subword tokenizers. It is available from GitHub (<https://github.com/google/sentencepiece>) and as a Python module with the command:

```
pip install sentencepiece
```

## 13.6 Hugging Face Tokenizers

The Hugging Face company has reimplemented a suite of tokenizers with the most common algorithms. They are easy to use and very fast. They come with pre-trained models and we can also train them on specific corpora.

The Hugging Face tokenizers share a common API in Python. A tokenizer consists of a pipeline of classes:

1. Normalizer, which normalizes the text, for instance in lowercase, by removing accents, or normalizing Unicode;
2. PreTokenizer, for instance a white space tokenizer;
3. Model, the algorithm to train from a corpus, BPE for instance;
4. Decoder, the program to tokenize a text once we a trained model;
5. PostProcessor, to add some special tokens such as start or end.

We can create a class implementing this sequence or reuse an existing one trained on a corpus. In the next section, we describe a pretrained BPE and then how to build a tokenizer from scratch.

### 13.6.1 Pretrained BPE

For this experiment, we will use the GPT-2 tokenizer. We create the model with the statements:

```
from tokenizers import Tokenizer
tokenizer = Tokenizer.from_pretrained('gpt2')
```

which downloads a `tokenizer.json` file containing the tokenizer parameters as well as the merge rules. This initializes `tokenizer` with the processing components, for instance, the pretokenizer. We can inspect these components:

```
>>> tokenizer.pre_tokenizer
<tokenizers.pre_tokenizers.ByteLevel at 0x7fb3f018c2f0>
>>> tokenizer.model
<tokenizers.models.BPE at 0x7ff3903efd10>
```

In the rest of this section, we show the tokenizer results with this quote:

```
ecloges_str = """Exiled from home am I; while, Tityrus, you
Sit careless in the shade"""
```

The result from the pretokenization is:

```
>>> tokenizer.pre_tokenizer.tokenize(ecloges_str)
[('Exiled', (0, 6)), ('from', (6, 11)), ('home', (11, 16)),
 ('am', (16, 19)), ...]
```

and for the whole tokenization:

```
>>> codes = tokenizer.encode(ecloges_str)
>>> codes.tokens
['Ex', 'iled', 'from', 'home', 'am', 'I', ';', 'while',
 ',', 'T', 'ity', 'rus', ',', 'you', 'G', 'C', 'Sit',
 'careless', 'Gin', 'the', 'Gshade']
```

### 13.6.2 Training BPE

We can also create a tokenizer from scratch and train it on a corpus. Here is a minimal configuration for BPE:

```
from tokenizers import Tokenizer, decoders, models, \
    pre_tokenizers, processors, trainers

tokenizer = Tokenizer(models.BPE())
tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space
    =False)
tokenizer.decoder = decoders.ByteLevel()
tokenizer.post_processor = processors.ByteLevel()
trainer = trainers.BpeTrainer(vocab_size=500)
```

We specify the vocabulary size to the trainer and we train it on Homer's *Iliad* and *Odyssey* stored in the `text` string. Before, we split the string into paragraphs:

```
trainer = trainers.BpeTrainer(vocab_size=500)
text_sentences = re.split(r'\n+', text)
tokenizer.train_from_iterator(text_sentences, trainer=trainer)
```

Once trained, we tokenize our quote:

```
>>> codes = tokenizer.encode(ecloges_str)
>>> codes.tokens
[‘E’, ‘x’, ‘ill’, ‘ed’, ‘from’, ‘home’, ‘Gam’, ‘GI’, ‘;’,
 ‘while’, ‘,’, ‘GT’, ‘ity’, ‘r’, ‘us’, ‘,’, ‘you’, ‘G’,
 ‘S’, ‘it’, ‘Gc’, ‘are’, ‘l’, ‘ess’, ‘Gin’, ‘the’, ‘Gsh’,
 ‘ade’]
```

and save the model:

```
tokenizer.save('homer.json')
```

## 13.7 Further Reading

While BPE dates from 1994, subword tokenizers are quite recent in natural language processing. Their adoption is due to the emergence of very large multilingual corpora and the impossibility to encode an unlimited number of words. This probably explains why their first applications were for machine translation; see Wu et al. (2016) *inter alia*. The transformers (Vaswani et al., 2017) popularized them as they are their standard input format. We will see this architecture in Chap. 15, *Self-Attention and Transformers*.

Subword tokenizers have many intricate technical details and, in doubt on their precise implementation, the best is to read their code. Karpathy (2022) provides a very didactical implementation of BPE and Kudo (2017) of SentencePiece, both on GitHub. Hugging Face created a fast implementation of many subword tokenizers in Rust,<sup>2</sup> as well as a good documentation of the API.<sup>3</sup>

---

<sup>2</sup> <https://github.com/huggingface/tokenizers/>.

<sup>3</sup> <https://huggingface.co/docs/tokenizers/index>.

# Chapter 14

## Part-of-Speech and Sequence Annotation



In Sect. 12.5, we saw that a same word may have two or more parts of speech leading to different morphological analyses or syntactic interpretations. The word *can* is an example of such an ambiguity as we can assign it two tags: either noun or modal verb. It is a noun in the phrase *a can of soup* and a modal verb in *I can swim*.

Ambiguity resolution, that is retaining only one part of speech (POS) and discarding the others, is generally referred to as POS tagging or POS annotation. In this chapter, we will see how to do this automatically with machine-learning techniques. We will proceed from the most simple to more elaborate systems. We will start with an elementary baseline technique, we will then use feed-forward networks with one-hot encoding and embeddings; we will finally use a new kind of device called recurrent neural networks.

### 14.1 Resolving Part-of-Speech Ambiguity

As children, we (possibly) learned to carry out manual grammatical analyses by considering the context of a word in a sentence and applying rules. In the phrase *a can of soup*, *can* is preceded by an article and therefore is part of a noun phrase. Since there is no other word before it, *can* is a noun. In the second phrase, *I can swim*, *can* is preceded by *I*, a pronoun, which would not precede a noun, and therefore *can* is a modal verb.

Voutilainen and Järvinen (1995) describe a more complex example with the sentence

That round table might collapse.

While the correct part-of-speech tagging is:

*That/determiner round/adjective table/noun might/modal verb collapse/verb.*

**Table 14.1** Ambiguities in part-of-speech annotation with the sentence: *That round table might collapse*

Words	Possible tags	Example of use	UPOS tags
<i>That</i>	Subordinating conjunction	<i>That he can swim is good</i>	SCONJ
	Determiner	<i>That white table</i>	DET
	Adverb	<i>It is not that easy</i>	ADV
	Pronoun	<i>That is the table</i>	PRON
	Relative pronoun	<i>The table that collapsed</i>	PRON
<i>round</i>	Verb	<i>Round up the usual suspects</i>	VERB
	Preposition	<i>Turn round the corner</i>	ADP
	Noun	<i>A big round</i>	NOUN
	Adjective	<i>A round box</i>	ADJ
	Adverb	<i>He went round</i>	ADV
<i>table</i>	Noun	<i>That white table</i>	NOUN
	Verb	<i>I table that</i>	VERB
<i>might</i>	Noun	<i>The might of the wind</i>	NOUN
	Modal verb	<i>She might come</i>	AUX
<i>collapse</i>	Noun	<i>The collapse of the empire</i>	NOUN
	Verb	<i>The empire can collapse</i>	VERB

a simple dictionary lookup or a morphological analysis produces many possible readings, as shown in Table 14.1.

As a first idea to create an automatic part-of-speech tagger, we can consider word preferences. Most words taken from a dictionary have only one part of speech or have a strong preference for only one of them, although frequent words tend to be more ambiguous. From text statistics based on different corpora, in English and in French, Merialdo (1994) and Vergne (1999) report that 50 to 60% of words have a unique possible tag, and 15 to 25% have only two tags. In both languages, tagging a word with its most common part of speech yields a success rate of more than 75%. Charniak (1993) even reports a score of more than 90% for English. This figure is called the **baseline**.

The baseline term is widely used in natural language processing to refer to a starting point that is usually easy to implement and to the score obtained with the corresponding minimal algorithm. We will then use the results obtained from the baseline to assess the improvements brought by more complex algorithms.

To create more accurate POS taggers, we can write logical rules mimicking human reasoning as in the introduction of this chapter or use machine-learning techniques, where we train a model on an annotated corpus. In the rest of the chapter, we will focus on machine learning.

## 14.2 Baseline

Before we start writing elaborate tagging algorithms, let us implement the baseline technique as it requires extremely limited efforts. As we saw, this baseline is to tag each word with its most frequent part of speech. We can derive the frequencies from a part-of-speech annotated corpus, such as those from the Universal Dependencies (UD) repository, see Sect. 12.3. Among the English UD corpora, the largest one is the English Web Treebank (EWT) with about 250,000 words split into training, validation, and test sets.

We collect the training, validation, and test sets from GitHub and we store them as strings in `train_sentences`, `val_sentences`, and `test_sentences`. Using the CoNLL dictorizer from Sect. 12.4), we create lists of dictionaries from these strings with the column names as keys:

```
conll_dict = CoNLLDictorizer(col_names)
train_dict = conll_dict.transform(train_sentences)
val_dict = conll_dict.transform(val_sentences)
test_dict = conll_dict.transform(test_sentences)
```

We count the parts of speech of such a formatted corpus with:

```
def count_word(corpus, word_key='FORM'):
    word_cnt = Counter()
    for sentence in corpus:
        for row in sentence:
            word_cnt[row[word_key]] += 1
    return word_cnt
```

We then compute the distribution of the parts of speech of a word with this function that we apply to the training set:

```
def distribution(corpus, word_key='FORM', pos_key='UPOS'):
    word_cnt = count_word(corpus, word_key)
    pos_dist = {key: Counter() for key in word_cnt.keys()}
    for sentence in corpus:
        for row in sentence:
            distribution = pos_dist[row[word_key]]
            distribution[row[pos_key]] += 1
    return pos_dist
```

As preprocessing, we can lowercase the letters or keep them as they are. Table 14.2 shows the distribution of the parts of speech for the words in the sentence *That round table might collapse* with counts extracted without changing the case.

The POS tagger model is then just a dictionary that associates a word with its most frequent part of speech. We use this function to obtain the associations:

```
word_pos = {}
for word in pos_dist:
    word_pos[word] = max(pos_dist[word],
                          key=pos_dist[word].get)
```

POS tagging is then very simple: It is just a dictionary lookup. Table 14.2 shows that on our highly ambiguous sentence, the baseline accuracy is only of 2/5 i.e. 40%.

**Table 14.2** Part of speech distribution in the EWT. We kept the original case of the letters

Words	Parts-of-speech counts	Most frequent POS	Correct POS
That	PRON: 58, DET: 15, SCONJ: 6	PRON	DET
round	NOUN: 4, ADV: 3, ADJ: 2, ADP: 2	NOUN	ADJ
table	NOUN: 14	NOUN	NOUN
might	AUX: 77	AUX	AUX
collapse	NOUN: 2, VERB: 1	NOUN	VERB

We cannot apply the tagger as is to the test set as it contains words unseen in the training set. A last thing is then to decide how to deal with such words. A first strategy is to use the most frequent tag. For this, we just need to apply a counter to the values of `word_pos`:

```
>>> Counter(word_pos.values())
Counter({'NOUN': 7322,
         'PROPN': 3805,
         'VERB': 3412,
         ...}
```

We would then tag the unseen words as nouns. Another idea is to check which tag would get the best results on a different corpus. We have the validation set for this that we will use to tune the model before we run the final evaluation on the test set. To implement this method, we compute the distribution of the unseen words, i.e. those not in the `word_pos` dictionary with:

```
def unseen_words_pos_dist(corpus,
                           word_pos,
                           word_key='FORM',
                           pos_key='UPOS'):
    unseen_words = Counter()
    for sentence in corpus:
        for word in sentence:
            if not word[word_key] in word_pos:
                unseen_words[word[pos_key]] += 1
    return unseen_words
```

And with apply this function to the validation set:

```
>>> unseen_words_pos_dist(val_dict, word_pos)
Counter({'PROPN': 739,
         'NOUN': 622,
         'VERB': 206,
         ...}
```

These results show that if we choose to tag an unknown word as a noun, we would be right 622 times, but 739 if we choose to tag it as a proper noun. This latter method is thus better and we will tag the unseen words as proper nouns.

**Table 14.3** Annotation of the English sentence: *I can't remember.* in the EWT. Note that *not* is annotated as a particle in this corpus rather than the more traditional adverb

ID	FORM	LEMMA	UPOS	FEATS
1	I	I	PRON	Case=Nom Number=Sing Person=1 PronType=Prs
2–3	can't	—	—	—
2	ca	can	AUX	VerbForm=Fin
3	n't	not	PART	—
4	remember	remember	VERB	VerbForm=Inf
5	.	.	PUNCT	—

## 14.3 Evaluation

Applying our tagger to the test set, we reach an accuracy of 86.4% on the EWT corpus for English and, with the same method, we obtain 91.2% on the GSD corpus (Guillaume et al. 2019) in French.

In the EWT and GSD corpora, contracted forms like *can't*, *don't*, or *won't* show twice in annotated sentences as in Table 14.3, with the original form, *can't*, and as two morphological parsed lemmas, *can* and *not*. Similarly in French, the words *du* and *des* are expanded respectively into *de le* and *de les*. We did not apply any specific preprocessing to remove them as this had insignificant consequences on the final accuracy.

In addition to accuracy, we can derive a confusion matrix that shows for each tag how many times a word has been correctly or wrongly labeled. Table 14.4 shows the results on the EWT test set for the baseline method. It enables us to understand and track errors. The diagonal shows the breakdown of the tags correctly assigned, for example, 96.8% for determiners (DET). The rest of the table shows the tags wrongly assigned, i.e., 1.7% of the determiners have been tagged as pronouns (PRON) and 0.9% as subordinating conjunctions (SCONJ). This table is only an excerpt, therefore the sum of rows is not equal to 100.

## 14.4 Part-of-Speech Tagging with Linear Classifiers

Linear classifiers, such as logistic regression or feed-forward neural networks, are an efficient set of numerical techniques we can use to carry out part-of-speech tagging. As input, the tagger reads the sentence's words sequentially from left to right and, using a model it has trained beforehand, predicts the part of speech of the current word.

To train and apply the model, the tagger extracts a set of features from the surrounding words, typically a sliding window spanning five words and centered on the current word. Core features are the lexical values of the words inside this

**Table 14.4** A confusion matrix breaking down the results of the baseline method on the EWT test set. The first column corresponds to the correct tags, and for each tag, the rows give the assigned tags. We set aside some parts of speech from the table. A perfect tagging would yield an identity matrix

Tagger →		Tagger →										
↓Correct		ADJ	ADP	ADV	AUX	CCONJ	DET	NOUN	PRON	PROPN	SCONJ	VERB
ADJ	82.8	0.6	1.5	0.	0.	0.1	1.9	0.	11.5	0.	1.6	
ADP	0.	88.2	0.4	0.	0.	0.	0.	0.	0.3	0.6	0.	
ADV	5.3	7.1	78.5	0.1	0.2	1.3	0.8	2.5	2.1	1.4	0.1	
AUX	0.	0.	88.9	0.	0.	0.1	0.1	0.1	0.3	0.	6.5	
CCONJ	0.	0.1	0.	0.	99.7	0.	0.	0.	0.1	0.	0.	
DET	0.2	0.	0.1	0.	0.2	96.8	0.1	1.7	0.1	0.9	0.	
NOUN	0.8	0.1	0.2	0.1	0.	76.2	0.	19.	0.1	3.1		
PRON	0.	0.	0.	0.	0.	2.1	0.	93.2	0.1	4.4	0.	
PROPN	1.1	0.3	0.	0.1	0.	0.	4.1	0.	93.6	0.	0.4	
SCONJ	0.	33.3	1.6	0.	0.	0.	0.	0.3	1.6	60.4	0.	
VERB	0.6	0.9	0.2	3.6	0.	0.	5.7	0.	7.2	0.	81.5	

**Fig. 14.1** Extracting features from the EWT to predict a part of speech. The features are extracted from a window of five words surrounding the word *visit*

ID	FORM	UPOS	
	BOS		Padding
	BOS		Padding
1	Or	CCONJ	
2	you	PRON	
3	can	AUX	
4	visit	VERB	<b>Predicted tag</b>
4	temples		↓
6	or		
7	shrines		
8	in		
9	Okinawa		
10	.		
	EOS		Padding
	EOS		Padding

ID	Feature vectors: $X$					UPOS: $y$
	$w_{i-2}$	$w_{i-1}$	$w_i$	$w_{i+1}$	$w_{i+2}$	
1	BOS	BOS	Or	you	can	CCONJ
2	BOS	Or	you	can	visit	PRON
3	Or	you	can	visit	temples	AUX
4	you	can	visit	temples	or	VERB
5	can	visit	temples	or	shrines	NOUN
6	visit	temples	or	shrines	in	CCONJ
7	temples	or	shrines	in	Okinawa	NOUN
8	or	shrines	in	Okinawa	.	ADP
9	shrines	in	Okinawa	.	EOS	PROPN
10	in	Okinawa	.	EOS	EOS	PUNCT

**Fig. 14.2** The feature vectors corresponding to the  $X$  input matrix and the parts of speech to predict representing the  $y$  output

window. They are produced by a tokenizer, possibly followed by a morphological parser.

To represent the dataset as a matrix, for a context window of five words, we associate the feature vectors  $\mathbf{x}_i = (w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2})$  with the part-of-speech tags  $y_i = t_i$  at index  $i$ . Using the sentence in Table 12.10, Fig. 14.1 shows an example of it centered on the word *visit*:

- The part of speech to predict is  $t_4 = \text{VERB}$ ;
- The surrounding words are  $w_2 = \text{you}$ ,  $w_3 = \text{can}$ ,  $w_4 = \text{visit}$ ,  $w_5 = \text{temples}$ , and  $w_6 = \text{or}$ .

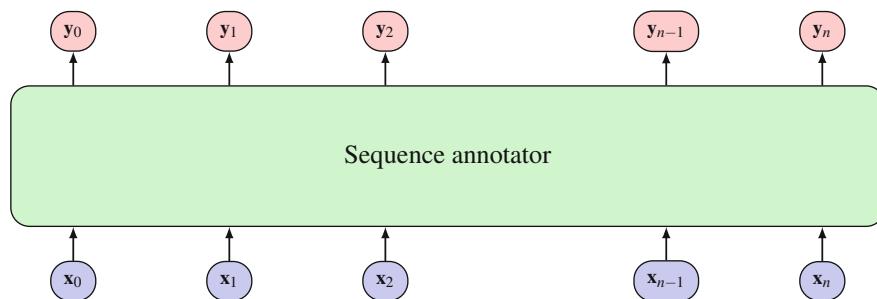
Figure 14.2 shows more feature vectors from this sentence. We first use this table to train a model; we then apply it sequentially to assign the tags. If we use logistic regression, the tagger outputs a probability,  $P(t_i | w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2})$ , that we can associate with each tag of the sequence.

At the beginning and end of the sentence, the window extends beyond the sentence boundaries. A practical way to handle this is to pad the sentence—the words and parts of speech—with dummy symbols such as BOS (beginning of sentence) and EOS (end of sentence) or `<s>` and `</s>`. If the window has a size of five words, we will pad the sentence with two BOS symbols in the beginning and two EOS symbols in the end.

We extract the features from POS-annotated corpora, and we train the models using machine-learning libraries such as scikit-learn (Pedregosa et al. 2011) for logistic regression or PyTorch for neural networks. Real systems would use more features than those from the core feature set such as the word characters, prefixes and suffixes, word bigrams, part-of-speech bigrams, etc. Nonetheless, the principles remain the same, the only difference would be a larger feature set.

## 14.5 Programming a Part-of-Speech Tagger with Logistic Regression

In this section, we will implement a simple tagger that uses a context of five words centered on the current word to predict the part of speech. We will first write a feature extractor, convert these features in numerical vectors, and store them in a matrix. We will then train and apply a logistic regression model. Figure 14.3 shows the overall structure of such a sequence annotator, where given an input  $\mathbf{x}_i$ , the system predicts an output  $\mathbf{y}_i$ . Note that we could apply this architecture to any kind of sequences, provided that they have equal length.



**Fig. 14.3** Sequence annotation, here using logistic regression and applied to part-of-speech tagging. The input and output sequences have equal length

### 14.5.1 Building the Feature Vectors

We load the training, validation, and test sets and convert them as lists of dictionaries as in Sects. 12.4 and 14.2. Let us name `train_dict`, `val_dict`, and `test_dict`, the datasets stored as lists of dictionaries.

Then, given a sentence, we extract the words before and after the current word. We store these words in a dictionary compatible with scikit-learn transformers so that we can vectorize them using the built-in `DictVectorizer()` class; see Sect. 6.4.1. To complete this extraction, we will proceed in two steps:

1. We extract the words and parts of speech from the dictionaries;
2. We create the word contexts as tables.

We extract the words and parts of speech of a sentence with the function:

```
def extract_cols(sent_dict, x='FORM', y='UPOS'):
    (input, target) = ([], [])
    for word in sent_dict:
        input += [word[x]]
        target += [word.get(y, None)]
    return input, target
```

We apply it to the full training, validation, and test sets:

```
train_cols = [extract_cols(sent_dict)
             for sent_dict in train_dict]
val_cols = [extract_cols(sent_dict)
            for sent_dict in val_dict]
test_cols = [extract_cols(sent_dict)
             for sent_dict in test_dict]
```

and we separate the words from the parts of speech with

```
train_sent_words, train_sent_pos = zip(*train_cols)
val_sent_words, val_sent_pos = zip(*val_cols)
test_sent_words, test_sent_pos = zip(*test_cols)
```

At this point, for the sentence in Table 12.10, we have the words:

```
>>> train_sent_words[8131]
['Or', 'you', 'can', 'visit', 'temples', 'or', 'shrines',
 'in', 'Okinawa', '.']
```

and the parts of speech:

```
>>> train_pos_words[8131]
['CCONJ', 'PRON', 'AUX', 'VERB', 'NOUN', 'CCONJ', 'NOUN',
 'ADP', 'PROPN', 'PUNCT']
```

Now, we can create the feature vectors consisting of words centered on the position of the part of speech to predict. The length of the feature lists is given by the left and right contexts of `w_size`. The function pads the sentence with begin-of-sentence (BOS) and end-of-sentence (EOS) symbols and extracts windows of  $2 \times w\_size + 1$  words as in Table 14.1:

```

def create_X_cat(sentence: list[str],
                 w_size: int = 2) -> list[dict[int: str]]:
    start_pads = ['__BOS__'] * w_size
    end_pads = ['__EOS__'] * w_size
    sentence = start_pads + sentence + end_pads
    # We extract the features
    X = []
    for i in range(len(sentence) - 2 * w_size):
        x = []
        for j in range(2 * w_size + 1):
            x += [sentence[i + j]]
        X += [x]
    X = [dict(enumerate(x)) for x in X]
    return X

```

Applying it to our sentence, we obtain a list of dictionaries, each dictionary being a sequence of feature values :

```

>>> create_X_cat(train_sent_words[8131])
[{:0: '__BOS__', 1: '__BOS__', 2: 'Or', 3: 'you', 4: 'can'},
{:0: '__BOS__', 1: 'Or', 2: 'you', 3: 'can', 4: 'visit'},
{:0: 'Or', 1: 'you', 2: 'can', 3: 'visit', 4: 'temples'},
{:0: 'you', 1: 'can', 2: 'visit', 3: 'temples', 4: 'or'},
{:0: 'can', 1: 'visit', 2: 'temples', 3: 'or', 4: 'shrines'},
{:0: 'visit', 1: 'temples', 2: 'or', 3: 'shrines', 4: 'in'},
{:0: 'temples', 1: 'or', 2: 'shrines', 3: 'in', 4: 'Okinawa'},
{:0: 'or', 1: 'shrines', 2: 'in', 3: 'Okinawa', 4: '.'},
{:0: 'shrines', 1: 'in', 2: 'Okinawa', 3: '.', 4: '__EOS__'},
{:0: 'in', 1: 'Okinawa', 2: '.', 3: '__EOS__', 4: '__EOS__'}]

```

matching the content in Fig. 14.2.

We create the  $X_{\text{cat}}$  matrix and  $y$  vector for the whole training set with:

```

X_train_cat = [row for sent in train_sent_words
               for row in create_X_cat(sent)]
y_train_cat = [pos for sent in train_sent_pos
               for pos in sent]

```

### 14.5.2 Encoding the Features

We vectorize the contexts stored in  $X_{\text{train\_cat}}$  with the scikit-learn built-in `DictVectorizer` class. As we saw in Sect. 6.4.1, `DictVectorizer` uses the `fit_transform()` function to fit the symbols to numbers and convert the list into a matrix:

```

from sklearn.feature_extraction import DictVectorizer

dict_vectorizer = DictVectorizer()
X_train = dict_vectorizer.fit_transform(X_train_cat)

```

The scikit-learn classifiers can handle  $y$  vectors consisting of symbols and we do not need to convert the POS classes into numbers.

### 14.5.3 Applying the Classifier

Now that we have the features in `X_train` and the parts of speech in `y_train_cat`, we can train a classifier. We use logistic regression from the `linear_model` module in scikit-learn and its `fit()` function:

```
from sklearn import linear_model
classifier = linear_model.LogisticRegression()
model = classifier.fit(X_train, y_train_cat)
```

We write a `predict()` function to carry out the POS prediction from a sentence. It encodes the features with the `transform()` function from `DictVectorizer`, where this time it does not need to fit the symbols, and applies the classifier with `predict()`. We store the predicted POS tags in the `ppos` key of the dataset dictionary:

```
def predict_sentence(sentence,
                     model,
                     dict_vectorizer,
                     ppos_key='PPOS'):
    sent_words, _ = extract_cols(sentence)
    X_cat = create_X_cat(sent_words)
    X = dict_vectorizer.transform(X_cat)
    y_pred_vec = model.predict(X)
    # We add the predictions in the PPOS column
    for row, y_pred in zip(sentence, y_pred_vec):
        row[ppos_key] = y_pred
    return sentence
```

and we apply it to all the sentences of the test set.

Using this simple program, where most of the code is to format the  $X$  matrix, we reach an accuracy of 90.24 on the English Web Treebank (EWT) (Silveira et al. 2014), an improvement of 3.8% over the baseline, and 94.05% on the French GSD corpus.

## 14.6 Part-of-Speech Tagging with Feed-Forward Networks

In this section, we will move from logistic regression to feed-forward neural networks and, as programming API, we will use PyTorch instead of scikit-learn.

### 14.6.1 Programming a Single Layer Network for POS Tagging

We load the dataset as in the previous section and the preprocessing to build the  $X$  matrix is almost identical. The only difference is to ensure that the PyTorch tensors

have the same numerical types as the NumPy arrays. For this, we vectorize the matrices as nonsparse and with a datatype of 32-bit floats:

```
dict_vectorizer = DictVectorizer(sparse=False, dtype=np.float32)
```

Once vectorized, we convert the resulting arrays to tensors with

```
X_train = torch.from_numpy(X_train)
X_val = torch.from_numpy(X_val)
X_test = torch.from_numpy(X_test)
```

The `y` vector is a vector of indices in PyTorch, while it can be a list of strings in scikit-learn. We first create conversion dictionaries:

```
idx2pos = dict(enumerate(sorted(set(y_train_cat))))
pos2idx = {v: k for k, v in idx2pos.items()}
```

and we convert the part-of-speech strings with:

```
y_train = torch.LongTensor(
    list(map(lambda x: pos2idx.get(x), y_train_cat)))
y_val = torch.LongTensor(
    list(map(lambda x: pos2idx.get(x), y_val_cat)))
y_test = torch.LongTensor(
    list(map(lambda x: pos2idx.get(x), y_test_cat)))
```

We create `TensorDataset` and `DataLoader` objects for our three sets. We set the `batch_size` to 512 for the training process meaning that will use 512 samples per update:

```
train_dataset = TensorDataset(X_train, y_train)
train_dataloader = DataLoader(
    train_dataset, batch_size=512, shuffle=True)

val_dataset = TensorDataset(X_val, y_val)
val_dataloader = DataLoader(
    val_dataset, batch_size=2048, shuffle=False)

test_dataset = TensorDataset(X_test, y_test)
test_dataloader = DataLoader(
    test_dataset, batch_size=2048, shuffle=False)
```

We use the `Sequential` module as in Sect. 8.5.3. The logistic regression model has just one linear layer. The input dimension is the length of the rows in `X` and the output dimension is given by the total number of parts of speech:

```
model = nn.Sequential(
    nn.Linear(X_train.size(dim=1), len(pos2idx))
)
```

and as optimizer, we use nadam with a learning rate `LR` of 0.005:

```
loss_fn = nn.CrossEntropyLoss()      # cross entropy loss
optimizer = torch.optim.NAdam(model.parameters(), lr=LR)
```

### 14.6.2 Training the Model

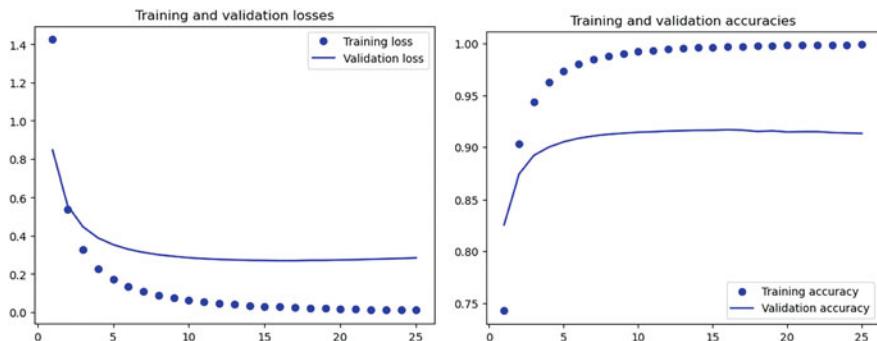
We can now write a training loop, where we fit the parameters on the training set and evaluate the results on the validation set. This will enable us to monitor the gradient descent and see when the model starts overfitting. To simplify the code, we define first an evaluation function that computes the loss and accuracy of a model. Its input parameters are the model, the loss function, and a `DataLoader` object:

```
def evaluate(model,
            loss_fn,
            dataloader) -> tuple[float, float]:
    model.eval()
    with torch.no_grad():
        loss = 0
        acc = 0
        batch_cnt = 0
        for X_batch, y_batch in dataloader:
            batch_cnt += 1
            y_batch_pred = model(X_batch)
            loss += loss_fn(y_batch_pred, y_batch).item()
            acc += (sum(torch.argmax(y_batch_pred, dim=-1)
                        == y_batch)/y_batch.size(dim=0)).item()
    return loss/batch_cnt, acc/batch_cnt
```

The training loop is similar to those we have already written. The training part is followed by an evaluation on the validation set:

```
for epoch in range(EPOCHS):
    train_loss = 0
    train_acc = 0
    batch_cnt = 0
    model.train()
    for X_batch, y_batch in tqdm(train_dataloader):
        batch_cnt += 1
        y_batch_pred = model(X_batch)
        loss = loss_fn(y_batch_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_acc += (sum(
            torch.argmax(y_batch_pred, dim=-1)
            == y_batch)/y_batch.size(dim=0)).item()
        train_loss += loss.item()
    model.eval()
    with torch.no_grad():
        history['accuracy'] += [train_acc/batch_cnt]
        history['loss'] += [train_loss/batch_cnt]
        val_loss, val_acc = evaluate(model, loss_fn, val_dataloader)
        history['val_loss'] += [val_loss]
        history['val_accuracy'] += [val_acc]
```

We plot the training and validation losses and accuracies curves to monitor if they follow the same trend. When the validation accuracy starts to decrease, this means



**Fig. 14.4** Feed-forward network with one-hot input. The training loss and accuracy over the epochs

that the model is overfitting and we should stop. Figure 14.4 shows the curves, where we observe an overfitting starting at epoch 17.

We apply the model trained on 16 epochs to the test set of the English Web Treebank and we compute the accuracy with:

```
evaluate(model, loss_fn, test_dataloader)
```

for which we reach a score of 91.62%.

Finally, we save the model with

```
torch.save(model.state_dict(), file_name)
```

### 14.6.3 Networks with Hidden Layers

Building a network with more layers is very easy with PyTorch. We just to add them to the model, where the activation function of the layers will be `ReLU`, except the last one.

As a rule of thumb, the number of nodes in hidden layers should gradually decrease from number of input nodes to the number of classes. Here we create a hidden layer with as twice as many nodes as there are parts of speech:

```
model = nn.Sequential(
    nn.Linear(X_train.size(dim=1), 2 * len(pos2idx)),
    nn.ReLU(),
    nn.Linear(2 * len(pos2idx), len(pos2idx)),
)
```

Using the same training set and hyperparameters identical to those of our previous experiment, the model starts overfitting at the second epoch. This architecture with one hidden layer does not improve the accuracy with a score of 91.43% after two epochs on the test set of the English Web Treebank. As a conclusion here, the simpler logistic regression model is more effective.

## 14.7 Embeddings

In the previous section, we encoded the words as one-hot vectors. In addition to producing very large matrices, such a representation does not take into account the possible proximity between the words. We will now use word embeddings instead: A dense representation that we introduced in Sect. 11.2, where semantically similar words are close in the vector space. As a consequence, and provided that the embedding vocabulary is large enough, the classifier will be able to relate words that are not in the training set to words present in the embedding table and integrate them in its model.

In our program, we will use GloVe (Pennington et al. 2014), a popular set of embeddings that we saw in Sect. 11.8. We can download pretrained GloVe embeddings from the internet.<sup>1</sup> The data is stored in files, where the words are represented by vectors of 25, 50, 100, 200, or 300 dimensions. In a file, each row consists of a word followed by its embedding vector.

Let us define a function to read an embedding file and return a dictionary, where the keys are the words and the values, the embedding vectors. In this example, we use GloVe.6B.100d.txt, a file of uncased 400,000 words represented as vectors of 100 dimensions. The code to read the file is the following:

```
def read_embeddings(file):
    embeddings = {}
    with open(file) as glove:
        for line in glove:
            values = line.strip().split()
            word = values[0]
            vector = torch.FloatTensor(
                list(map(float, values[1:])))
            embeddings[word] = vector
    return embeddings

>>> embeddings_dict = read_embeddings(embedding_file)
```

For example, the first values of the 100-dimensional embedding vector representing the word *table* are:

(−0.615, 0.897, 0.568, 0.391, −0.224, 0.490, 0.109, 0.274, −0.238, −0.522, ...)

To use an embedding layer in a PyTorch classifier, we need to build an index of unique words that covers the embedded words as well as the words seen in the training corpus: The embedded words are just the keys of `embeddings_dict` and the words of the corpus correspond to the words in the contexts we built in Sect. 14.5.1. We first extract all the values from all the contexts, we lowercase them as GloVe is uncased, and we build a set of them:

---

<sup>1</sup> <https://nlp.stanford.edu/projects/glove/>.

```

corpus_words = [value.lower() for x in X_train_cat
                for value in x.values()]
corpus_words = sorted(set(corpus_words))

```

To create the index, we merge the words in the training corpus with the words in the GloVe file and we create a set of all the words.

```

embeddings_words = embeddings_dict.keys()
vocabulary = set(corpus_words + list(embeddings_words))

```

We then create a word index, where we keep index 0 for the unknown words:

```

idx2word = dict(enumerate(sorted(vocabulary), start=1))
word2idx = {v: k for k, v in idx2word.items()}

```

Finally, we replace the lowercased words with their index in the context dictionaries:

```

for x_train_cat in X_train_cat:
    for word in x_train_cat:
        x_train_cat[word] = word2idx[
            x_train_cat[word].lower()]

for x_val_cat in X_val_cat:
    for word in x_val_cat:
        x_val_cat[word] = word2idx.get(
            x_val_cat[word].lower(), 0)

for x_test_cat in X_test_cat:
    for word in x_test_cat:
        x_test_cat[word] = word2idx.get(
            x_test_cat[word].lower(), 0)

```

We apply `DictVectorizer()` to transform the dictionaries containing the training, validation, and test sets into matrices:

```

dict_vectorizer = DictVectorizer(dtype=np.int64, sparse=False)
X_train = dict_vectorizer.fit_transform(X_train_cat)
X_val = dict_vectorizer.transform(X_val_cat)
X_test = dict_vectorizer.transform(X_test_cat)

```

Note that the resulting matrices contain indices and not one-hot vectors. We process the parts of speech the same way as we did in Sect. 14.6.1.

We create the embedding table with a random initialization

```

embedding_table = torch.randn(
    (len(vocabulary) + 1, EMBEDDING_DIM))/10

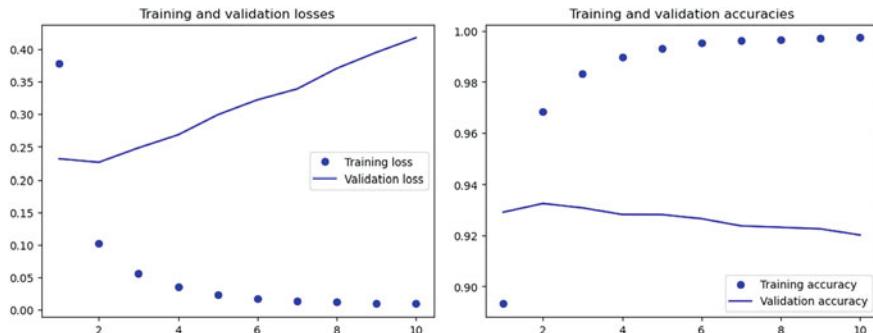
```

and we fill the index values when available with the GloVe embeddings:

```

for word in vocabulary:
    if word in embeddings_dict:
        embedding_table[word2idx[word]] = embeddings_dict[word]

```



**Fig. 14.5** Feed-forward network with embedding input. The training loss and accuracy over the epochs

Now all is in place to create the neural network. We use a sequential model: It consists of a trainable embedding layer that fetches the embeddings from the word indices in the  $X$  matrix storing the contexts; we load the 100-dimension GloVe embeddings with `from_pretrained()` as in Sect. 11.8; a `Flatten()` class that flattens the embedded vectors; and a linear layer:

```
model = nn.Sequential(
    nn.Embedding.from_pretrained(
        embedding_table, freeze=False),
    nn.Flatten(),
    nn.Linear(5 * embedding_table.size(dim=1),
              len(pos2idx))
)
```

The rest of the code and hyperparameters is identical to that of the previous section. We also use the NAdam optimizer with a learning rate of 0.005.

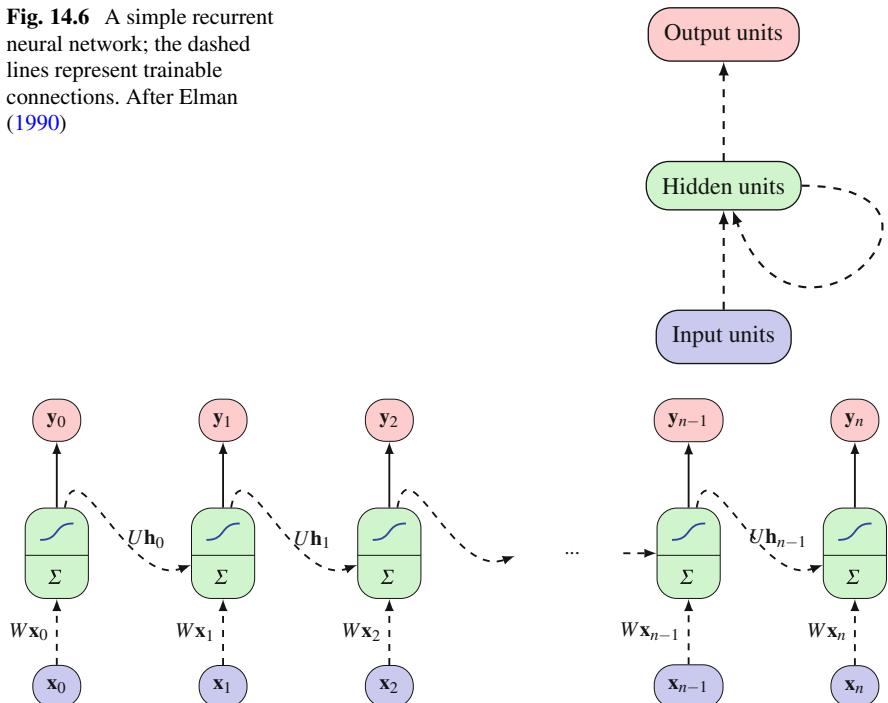
Figure 14.5 shows the loss and accuracy curves. We see that the model starts overfitting at epoch 3. After training the model on two epochs on the training set of the English Web Treebank, we obtain an accuracy of 93.34 on the test set. This is nearly 2% better than with a one-hot encoding.

## 14.8 Recurrent Neural Networks

In Figs. 14.1 and 14.2, we predicted a part of speech  $t_i$  from the words surrounding it,  $(w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2})$ . Once identified, this part of speech certainly sets constraints on the next one, for instance a determiner will probably exclude a verb as following tag. A possible improvement to our previous programs would be to include one or two previous predictions,  $t_{i-1}$  and  $t_{i-2}$  in the feature vector so that we have more information on the context:

$$P(t_i | w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1}).$$

**Fig. 14.6** A simple recurrent neural network; the dashed lines represent trainable connections. After Elman (1990)



**Fig. 14.7** The network from Fig 14.6 unfolded in time. In this figure, the connection between the hidden states and the outputs are not trainable

Although we could implement such features with a feed-forward architecture, the recurrent neural networks (RNN) are a class of networks that are just designed for that: Reuse the prediction from the previous step (the output) as input to the current one.

Elman (1990) proposed a simple recurrent structure with three layers consisting of input, hidden, and output units. Such a structure is very similar to a feed-forward network with one hidden layer. The only difference is that the output from the hidden units is reintroduced as input to the next step of the sequence, see Fig. 14.6. We can spread the cells through the input sequence to understand how the network handles each input. Figure 14.7 shows the unfolded units of a simple RNN model.

We use two matrices to model the two trainable connections: One between the input and the hidden layer,  $W$ , and a second one,  $U$ , between two consecutive hidden units of the sequence. We compute the values in the hidden layers at index  $i$  in the sequence with these matrices and a recurrent formula that includes a possible intercept,  $\mathbf{b}$ :

$$W\mathbf{x}_i + Uh_{i-1} + \mathbf{b}.$$

The number of rows in  $W$  corresponds to the number of hidden units. We pass the result through an activation function, usually a hyperbolic tangent, to get the output of the hidden layer at index  $i$ :

$$\mathbf{h}_i = \tanh(W\mathbf{x}_i + U\mathbf{h}_{i-1} + \mathbf{b}).$$

The hidden vector  $\mathbf{h}_i$  is passed to a linear layer and then a softmax function to predict the output vector  $\mathbf{y}_i$ .

### 14.8.1 Programming RNNs for POS Tagging

The design of a simple part-of-speech tagger with recurrent neural networks and PyTorch is easy. We load the datasets and we dictorize them as in Sect. 14.5.1. We then extract the  $\mathbf{x}$  and  $\mathbf{y}$  aligned sequences from all the sentences to create the  $X$  and  $Y$  matrices. We split the matrix construction into four steps:

1. For each sentence in the corpus, collect the parallel sequences of words and parts of speech. This corresponds to the FORM and UPOS columns in Table 12.10;
2. Build the word and part-of-speech indices and replace each word and part of speech in the lists with their index;
3. Pad the lists with a padding symbol so that all the lists have the same length and create two matrices from the two lists of lists;
4. Finally, convert the numbers into embedding vectors. As in Sect. 14.7, we will use the GloVe embeddings.

We collect the lists of words and parts of speech using the `extract_cols()` function as in Sect. 14.5.1. It extracts by default the FORM and UPOS columns and returns two lists per sentence. Using the same comprehension as in Sect. 14.5.1, we build the lists of words and parts of speech:

```
train_sent_words, train_sent_pos
val_sent_words, val_sent_pos
test_sent_words, test_sent_pos
```

For the training set, we run the statement

```
train_sent_words = [list(map(str.lower, sentence))
                   for sentence in train_sent_words]
```

where we set the words in lowercase to be compatible with GloVe. We do the same with the validation and test sets.

## Indices

To create the word and part-of-speech indices, we first collect the list of unique words as well as the parts of speech from the training set.

```

corpus_words = sorted(set([word
                           for sentence in train_sent_words
                           for word in sentence]))

pos_list = sorted(set([pos
                           for sentence in train_sent_pos
                           for pos in sentence]))

```

We also collect the words from the embedding dictionary as in Sect. 14.7 and we merge the vocabularies:

```

embeddings_words = embeddings_dict.keys()
vocabulary = set(corpus_words + list(embeddings_words))

```

We then create the word indices that will serve as input to the PyTorch embedding table.

Sequences of words or characters are by nature of different lengths. As PyTorch processes mostly tensors of rectangular dimensions, we need to pad the input so that all the sequences have an equal length. To make a distinction with other indices, the `Embedding` class has an argument that assigns the padding symbol to a given index: `padding_idx`. In the embedding table, the corresponding row will be filled with zeros and the parameters will be nontrainable.

```

>>> embedding_pad = nn.Embedding(vocab_size, embedding_dim,
                                 padding_idx=0)

```

In the word index, we start at 2 to make provision for the padding symbol 0 and unknown words, 1. The POS index will start at 1 to differentiate the POS indices with the padding symbol too:

```

idx2word = dict(enumerate(vocabulary, start=2))
idx2pos = dict(enumerate(pos_list, start=1))
word2idx = {v: k for k, v in idx2word.items()}
pos2idx = {v: k for k, v in idx2pos.items()}

```

We create the parallel sequences of indexes with a `to_index()` function that carries out a lookup in the index dictionary:

```

def to_index(X: list[list[str]],
            word2idx: dict[str, int],
            unk: int = 1) -> torch.LongTensor:
    X_idx = []
    for x in X:
        # We map the unknown words to one
        x_idx = torch.LongTensor(
            list(map(lambda x:
                     word2idx.get(x, unk), x)))
        X_idx += [x_idx]
    return X_idx

```

We convert the lists of training sentences and parts of speech into tensors:

```

X_train_idx = to_index(train_sent_words, word2idx)
Y_train_idx = to_index(train_sent_pos, pos2idx)

```

We apply the same procedure to `X_val_idx`, `Y_val_idx`, `X_test_idx`, and `Y_test_idx`.

At this point, we have converted our datasets into parallel lists of word and part-of-speech indices.

## Padding

We can now pad the sequences to an identical length with the PyTorch `pad_sequence()` built-in function. This length corresponds to that of the longest sentence in the batch. As we decided, we set the padding index to 0.

We saw earlier that the traditional format of the  $X$  input matrix is to store the features of an observation, here a sentence, in a row. Each row containing the words of this sentence. We then stack vertically the observations in  $X$ . The first dimension of the input matrix is then called the batch dimension or batch axis.

PyTorch has a default format for the recurrent networks and the `pad_sequence()` function, where the columns are the sentences and the rows, the words across the sentences. For instance, the first row contains the first word of all the sentences in the batch. It is possible to change this order with the argument `batch_first=True` that restores the traditional order.

To see the difference, let us pad three short sentences in the corpus with the traditional order:

```
>>> pad_sequence(X_train_idx[8131:8134], batch_first=True)

tensor([[272459, 396470, 91221, 381705, 357368, 272459,
        331885, 189667, 270325,      925],
       [387146, 149841, 396470, 91221, 368115, 340721,
        196974,      925,          0,          0],
       [164208, 362757, 43521, 349110, 364861,      925,
        0,          0,          0,          0]])
```

and with the default:

```
>>> pad_sequence(X_train_idx[8131:8134])

tensor([[272459, 387146, 164208],
       [396470, 149841, 362757],
       [91221, 396470, 43521],
       [381705, 91221, 349110],
       [357368, 368115, 364861],
       [272459, 340721,      925],
       [331885, 196974,          0],
       [189667,      925,          0],
       [270325,          0,          0],
       [      925,          0,          0]])
```

We will use the traditional batch-first order in the rest of this chapter as we are used to it.

```
from torch.nn.utils.rnn import pad_sequence

X_train = pad_sequence(X_train_idx, batch_first=True)
Y_train = pad_sequence(Y_train_idx, batch_first=True)
```

We create similarly `X_val`, `Y_val`, `X_test`, and `Y_test`.

At this point, we have represented our datasets as tensors of indices for the words and the parts of speech.

## Embeddings

We create the embedding matrix with the values from GloVe or random values for the words in the training corpus, but not in GloVe:

```
EMBEDDING_DIM = 100

embedding_table = torch.randn(
    (len(vocabulary) + 2, EMBEDDING_DIM))/10
for word in vocabulary:
    if word in embeddings_dict:
        embedding_table[word2idx[word]] = embeddings_dict[word]
```

## Network Architecture

Now we have processed the data so that we can use it as input to a RNN. We build the network with a class derived from `Module`, where we stack layers in a pipeline. We start with an embedding layer; we add a PyTorch `RNN()` layer; and finally a `Linear()` layer to make the prediction. The most important parameters of `RNN()` are:

1. The input size, that corresponds here to the embedding dimension;
2. The hidden size, that corresponds to the output dimension of the recurrent network. We use here 128;
3. We define the number of recurrent layers that we want to stack with `num_layers`. We will create two layers;
4. We set `batch_first` to true;
5. We may define a `dropout` rate that we will explain in a next section;
6. By default, `RNN()` runs the cells from left to right as shown in Fig. 14.7. It does not take into account the RNN outputs to the right of the current word. We set the `bidirectional` Boolean to true to run `RNN()` from left to right and right to left in parallel. Then for a given word, a bidirectional RNN layer concatenates the left and right RNN outputs.

In a bidirectional network, we have twice the number of hidden nodes of a single direction. We need then to multiply the linear input size by two.

As output, a RNN object returns two values: The whole sequence of predictions and the last prediction corresponding to the last word in the sentence. We will predict the whole sequence of parts of speech. Our class is then:

```
class Model(nn.Module):

    def __init__(self, embedding_table, hidden_size,
                 nbr_classes, freeze_embs=True,
                 num_layers=1, bidirectional=False):
        super().__init__()
        embedding_dim = embedding_table.size(dim=-1)
        self.embeddings = nn.Embedding.from_pretrained(
            embedding_table,
            freeze=freeze_embs,
            padding_idx=0)
        self.recurrent = nn.RNN(embedding_dim,
                               hidden_size,
                               batch_first=True,
                               num_layers=num_layers,
                               bidirectional=bidirectional)
        if not bidirectional:
            self.fc = nn.Linear(hidden_size, nbr_classes)
        else:
            # twice the units if bidirectional
            self.fc = nn.Linear(2*hidden_size, nbr_classes)

    def forward(self, sentence):
        embeds = self.embeddings(sentence)
        rec_out, _ = self.recurrent(embeds)
        logits = self.fc(rec_out)
        return logits
```

We create a model with:

```
model = Model(embedding_table,
              128,
              len(pos2idx) + 1,
              freeze_embs=False,
              num_layers=2,
              bidirectional=True)
```

where we add 1 to the number of classes to take the padding symbol into account.

We define a loss and an optimizer, where we tell the loss to ignore the padding index:

```
loss_fn = nn.CrossEntropyLoss(ignore_index=0)
optimizer = torch.optim.NAdam(model.parameters(), lr=0.005)
```

## Training Loop

The training loop is overall identical to those we have already written. The only difference is in the computation of the loss. Our input consists of sentences vertically

stacked in a matrix and the model outputs sequences of predictions. This output is a 3rd order tensor (sometimes called a 3D matrix), where the first axis is the batch, the second one, the word position in the sentence, and the 3rd one, the probabilities of the parts of speech.

As input to the loss function, we need to provide a tensor of probabilities and a tensor of indices. To do this, we flatten the  $Y$  matrix containing the indices of the true parts of speech in one single vector of indices. We use: `Y.reshape(-1)`. Similarly,  $Y_{\text{pred}}$  is the stack of sentences, where the rows contain the probabilities of the parts of speech for each input word. We reshape them in a matrix, where the first axis represents the rows put end to end and the second axis, the prediction probabilities for each part of speech:

```
Y_pred.reshape(-1, Y_pred.size(dim=-1))
```

We create `Dataset` and `DataLoader` objects for the training set:

```
train_dataset = TensorDataset(X_train, Y_train)
train_dataloader = DataLoader(
    train_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

as well as for the validation and test sets.

The evaluation function is more complex than for the feed-forward networks as we have to remove the padding symbols from the computation:

```
def evaluate(model,
            loss_fn,
            dataloader) -> tuple[float, float]:
    model.eval()
    with torch.no_grad():
        loss = 0
        accuracy = 0
        t_words = 0
        for X_batch, Y_batch in dataloader:
            Y_batch_pred = model(X_batch)
            loss = loss_fn(
                Y_batch_pred.reshape(
                    -1,
                    Y_batch_pred.size(dim=-1)),
                Y_batch.reshape(-1))

            n_words = torch.sum(Y_batch > 0).item()
            t_words += n_words
            loss += n_words * loss.item()
            accuracy += torch.mul(
                torch.argmax(Y_batch_pred, dim=-1) == Y_batch,
                Y_batch > 0).sum().item()
    return loss/t_words, accuracy/t_words
```

We integrate the evaluation and computation of the loss in the training loop:

```

for epoch in range(EPOCHS):
    train_loss = 0
    train_accuracy = 0
    t_words = 0
    model.train()
    for X_batch, Y_batch in tqdm(train_dataloader):
        Y_batch_pred = model(X_batch)
        loss = loss_fn(
            Y_batch_pred.reshape(
                -1,
                Y_batch_pred.size(dim=-1)),
            Y_batch.reshape(-1))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        with torch.no_grad():
            n_words = torch.sum(Y_batch > 0).item()
            t_words += n_words
            train_loss += n_words * loss.item()
            train_accuracy += torch.mul(
                torch.argmax(
                    Y_batch_pred, dim=-1) == Y_batch,
                Y_batch > 0).sum().item()

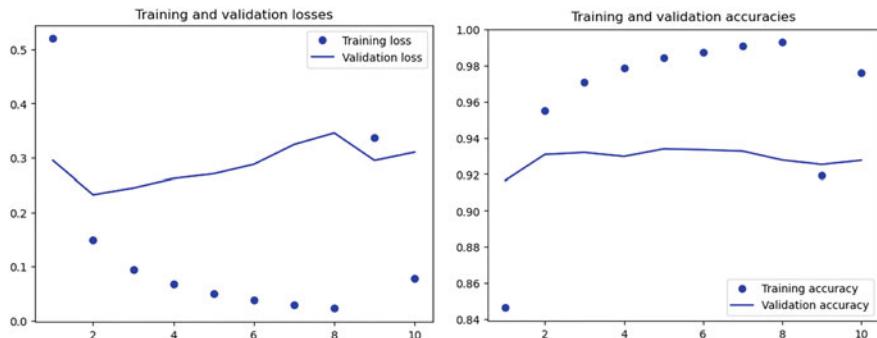
    model.eval()
    with torch.no_grad():
        history['accuracy'] += [
            train_accuracy/t_words]
        history['loss'] += [train_loss/t_words]
        val_loss, val_acc = evaluate(model,
                                      loss_fn,
                                      val_dataloader)
        history['val_loss'] += [val_loss]
        history['val_accuracy'] += [val_acc]

```

With a batch size of 128 and hyperparameters identical to our previous experiments for the rest, the model starts overfitting at epoch 3. We reach an accuracy of 93.47% on the test set of the English Web Treebank after two epochs. This is slightly better than the feed-forward network with embeddings.

### 14.8.2 Dropout

Compared to logistic regression, neural networks have a more complex architecture that enables them to model associations in the data. The downside of this is that they can overfit the training set, i.e. the models are able to classify perfectly the samples they have seen, but have a much lower performance on the test set or any unseen sample.



**Fig. 14.8** RNN network with embedding input. The training loss and accuracy over the epochs

One way to fight the overfit of neural networks is to drop out some connections, that is set them to zero, randomly at each update in the training step. In PyTorch, we can insert dropout layers anywhere in a sequential architecture using this statement:

```
nn.Dropout(rate)
```

where `rate` is a number between 0 and 1 representing the fraction of dropped out inputs: 0, no input is dropped, 1, all the inputs are dropped. This dropout is only for the training step and we must be careful to set the model in the evaluation mode for inferences:

```
model.eval()
```

For RNN layers, we set the corresponding dropout rates with the `dropout` parameter of the `RNN` class.

Figure 14.8 shows the accuracy and the loss using the previous hyperparameters and a dropout rate of 0.2. We reach an accuracy of 93.60% on the test set of the English Web Treebank after 2 epochs. This is about the same as without dropout, but the training curves are more regular.

### 14.8.3 LSTM

As we have seen, simple RNNs use the previous output as input. They have then a very limited feature context. Although, we can run them in both directions, they cannot memorize large word windows and model long dependencies.

Long short-term memory units (LSTM) (Hochreiter and Schmidhuber 1997) are an extension to RNNs that can remember, possibly forget, information from longer or more distant sequences. This has made them a popular technique for part-of-speech tagging.

Given an input at index  $t$ ,  $\mathbf{x}_t$ , a LSTM unit produces:

- A short term state, called  $\mathbf{h}_t$  and
- A long-term state, called  $\mathbf{c}_t$  or memory cell.

We use the short-term state,  $\mathbf{h}_t$ , to predict the unit output, i.e.  $\mathbf{y}_t$ ; both the long-term and short-term states serve as inputs to the next unit.

A LSTM unit starts from a core equation that is identical to that of a RNN. It uses two matrices and computes the dot product with, respectively, the input,  $\mathbf{x}_t$ , and the previous output,  $\mathbf{h}_{t-1}$ . The result is passed through a hyperbolic tangent activation function:

$$\mathbf{g}_t = \tanh(W_g \mathbf{x}_t + U_g \mathbf{h}_{t-1} + \mathbf{b}_g).$$

From the previous output and current input, we compute three kinds of filters, or gates, that will control how much information is passed through the LSTM cell. Each of these gates uses two matrices, a bias, and have the form of a RNN equation. The two first gates,  $\mathbf{i}$  and  $\mathbf{f}$ , defined as:

$$\begin{aligned}\mathbf{i}_t &= \text{activation}(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + \mathbf{b}_i), \\ \mathbf{f}_t &= \text{activation}(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + \mathbf{b}_f),\end{aligned}$$

model respectively how much we will keep from the base equation and how much we will forget from the long-term state.

To implement this selective memory, we apply the two gates to the base equation and to the previous long-term state with the element-wise product (Hadamard product), denoted  $\odot$ , and we sum the resulting terms to get the current long-term state:

$$\mathbf{c}_t = \mathbf{i}_t \odot \mathbf{g}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1}.$$

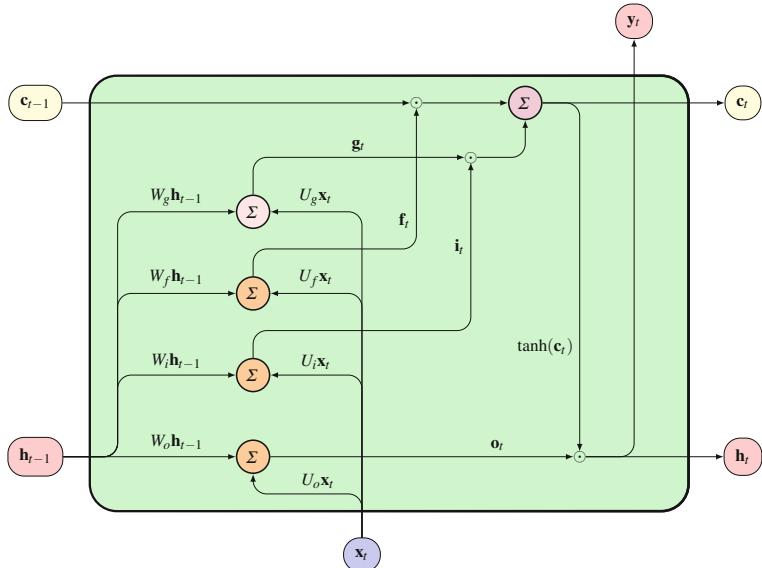
The third gate:

$$\mathbf{o}_t = \text{activation}(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o)$$

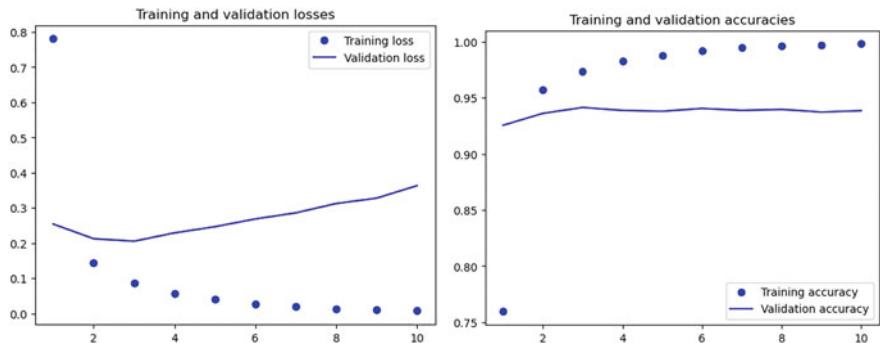
modulates the current long-term state to produce the output:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t).$$

Figure 14.9 shows the data flow between the inputs,  $\mathbf{c}_{t-1}$  and  $\mathbf{h}_{t-1}$ , and the outputs  $\mathbf{c}_t$  and  $\mathbf{h}_t$ . As for the other forms of neural networks, the LSTM parameters are determined by a gradient descent.



**Fig. 14.9** An LSTM unit showing the data flow, where  $g_t$  is the unit input,  $i_t$ , the input gate,  $f_t$ , the forget gate, and  $o_t$ , the output gate. The activation functions have been omitted



**Fig. 14.10** LSTM network with embedding input. The training loss and accuracy over the epochs

#### 14.8.4 POS Tagging with LSTMs

Implementing a LSTM tagger with PyTorch is identical to a RNN one. We just replace the `RNN` class with `LSTM` in the definition of our model. Figure 14.10 shows the training and validation curves. We reach an accuracy of 94.35% on the test set of the English Web Treebank after 3 epochs. This is about 0.75% better than with a simple RNN.

### 14.8.5 Further Improvements

In our experiments, starting from a baseline technique with a 86.4% accuracy and going through different architectures, we gradually increased the tagging performance to finally reach the figure of 94.35% with a LSTM.

This is not the end of the road however. There many ways to optimize further the systems we have described. One technique is to find better hyperparameters by tuning the batch size, number of nodes, dropout rate, the optimizer, its learning rate, etc. This is a tedious task, but it can lead to a considerable increase in performance. Another strategy is to modify the architecture by adding more layers: ReLU, dropout, linear, or recurrent.

We can also try to reduce the training and inference times. One shortcut we used to have rectangular matrices was to pad the whole dataset. This is far from optimal as the matrices then contain many useless symbols. This certainly slows down training and evaluation. In addition, it only works here because the dataset is relatively small and we do not have very long sentences. A better procedure would be to gather the sentences of same length into bins, create batches from these bins, and thus avoid padding or keep it to a minimum.

Finally, we could add a layer to check the consistency of the output sequence. Conditional random fields is a technique that would enable us to do that. We will describe it in Sect. 14.12.

## 14.9 Named Entity Recognition and Chunking

In Sect. 14.8, we saw that the LSTM networks enabled us to obtain the best results in part-of-speech annotation. In fact, these techniques can apply to any sequence of symbols, either as input or output. In this section, we will describe application examples with partial syntactic groups and named entities and we will start with a few definitions.

### 14.9.1 Noun Groups and Verb Groups

We saw in Sect. 12.1.3 that the two major parts of speech are the noun and the verb. In a sentence, nouns and verbs are often connected to other words that modify their meaning, for instance an adjective or a determiner for a noun; an auxiliary for a verb. These closely connected words correspond to partial syntactic structures, or groups, called **noun groups**, when the most important word in the structure is a noun or **verb groups**, when it is a verb. The terms **noun chunks** and **verb chunks** are also widely used and their identification in a sentence is called **chunking**.

**Table 14.5** Noun groups

English	French	German
<i>The waiter is bringing the very big dish to the table</i>	<i>Le serveur apporte le très grand plat sur la table</i>	<i>Der Ober bringt die sehr große Speise an den Tisch</i>
<i>Charlotte has eaten the meal of the day</i>	<i>Charlotte a mangé le plat du jour</i>	<i>Charlotte hat die Tagesspeise gegessen</i>

**Table 14.6** Verb groups

English	French	German
<i>The waiter is bringing the very big dish to the table</i>	<i>Le serveur apporte le très grand plat sur la table</i>	<i>Der Ober bringt die sehr große Speise an den Tisch</i>
<i>Charlotte has eaten the meal of the day</i>	<i>Charlotte a mangé le plat du jour</i>	<i>Charlotte hat die Tagesspeise gegessen</i>

More formally, noun groups (Table 14.5) and verb groups (Table 14.6) correspond to verbs and nouns and their immediate depending words. This is often understood, although not always, as words extending from the beginning of the constituent to the head noun or the head verb. That is, the groups include the headword and its dependents to the left. They exclude the postmodifiers. For the noun groups, this means that modifying prepositional phrases or, in French, adjectives to the right of the nouns are not part of the groups.

The principles we exposed above are very general, and exact definitions of groups may vary in the literature. They reflect different linguistic viewpoints that may coexist or compete. However, precise definitions are of primary importance. Like for part-of-speech tagging, hand-annotated corpora will solve the problem. Most corpora come with annotation guidelines. They are usually written before the hand-annotation process. As definitions are often difficult to formulate the first time, they are frequently modified or complemented during the annotation process. Guidelines normally contain definitions of groups and examples of them. They should be precise enough to enable the annotators to identify consistently the groups. The annotated texts will then encapsulate the linguistic knowledge about groups and make it accessible to the machine-learning techniques.

### 14.9.2 Named Entities

In the NLP literature, a named entity is a term closely related to that of a proper noun. The word entity is roughly a synonym for object, thing and a named entity is an entity whose name in a text refers to a unique person, place, object, etc., as *William Shakespeare* or *Stratford-upon-Avon* in the phrase:

William Shakespeare was born and brought up in Stratford-upon-Avon.

We can also say that *William Shakespeare* or *Stratford-upon-Avon* are mentions of named entities in the sentence.

This opposes to phrases referring to entities with no name as *this person* or *a street* in:

meeting with this person in a street nearby,

Again this reflects overall the distinction between common and proper nouns; see Fig. 14.11.

Names of people or organizations are frequent in the press and the media, where they surge and often disappear quickly. The first step before any further processing is to identify the phrases corresponding to names of persons, organizations, or locations (Table 14.7). Such phrases can be a single proper noun or a group of words.

Named entity recognition also commonly extends to temporal expressions describing times and dates, and numerical and quantity expressions, even if these are not entities.

## 14.10 Group Annotation Using Tags

The most intuitive way to mark the groups, noun groups, verb groups, or named entities, is to bracket them with opening and closing parentheses. Ramshaw and Marcus (1995) give examples below of noun group bracketing, where they insert brackets between the words where appropriate:

[<sub>NG</sub> The government <sub>NG</sub>] has [<sub>NG</sub> other agencies and instruments <sub>NG</sub>] for pursuing [<sub>NG</sub> these other objectives <sub>NG</sub>].

Even [<sub>NG</sub> Mao Tse-tung <sub>NG</sub>] [<sub>NG</sub> 's China <sub>NG</sub>] began in [<sub>NG</sub> 1949 <sub>NG</sub>] with [<sub>NG</sub> a partnership <sub>NG</sub>] between [<sub>NG</sub> the communists <sub>NG</sub>] and [<sub>NG</sub> a number <sub>NG</sub>] of [<sub>NG</sub> smaller, non-communists parties <sub>NG</sub>].

We can recast this bracketing procedure as a sequence annotation. Using tags, group annotation will then be very similar to part-of-speech tagging and we will be able to use the same techniques and algorithms.

For the sake of simplicity, we first present annotation schemes for noun groups. We will then generalize them to verb groups and other groups.

### 14.10.1 The IOB Tagset

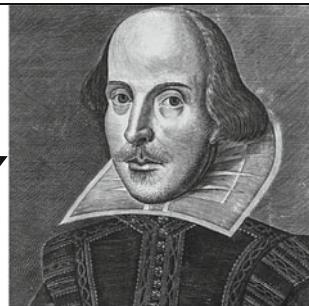
Ramshaw and Marcus (1995) defined a tagset of three elements {I, O, B}, where I means that the word is inside a noun group, O means that the word is outside, and B (between) means that the word is at the beginning of a noun group that immediately

---

**Named entities**

---

William Shakespeare was born and brought  
up in Stratford-upon-Avon

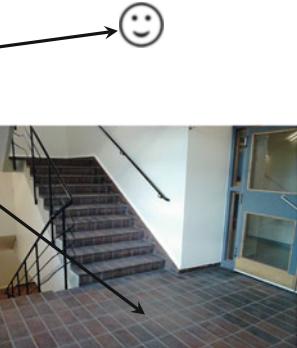


---

**Other entities**

---

Meeting with our guest on the landing at  
lunchtime



**Fig. 14.11** Named entities: entities that we can identify by their names. Portrait: credits Wikipedia. Map: Samuel Lewis, *Atlas to the topographical dictionaries of England and Wales*, 1848, credits: archive.org

**Table 14.7** Named entities in English and French

Type	English	French
Company names	<i>British Gas plc.</i>	<i>Compagnie générale d'électricité SA</i>
Person names	<i>Mr. Smith</i>	<i>M. Dupont</i>
Titles	<i>The President of the United States</i>	<i>Le président de la République</i>

follows another noun group. Using this tagging scheme, an equivalent annotation of the sentences above is:

The/I government/I has/O other/I agencies/I and/I instruments/I for/O pursuing/O these/I other/I objectives/I ./O

Even/O Mao/I Tse-tung/I 's/B China/I began/O in/O 1949/I with/O a/I partnership/I between/O the/I communists/I and/O a/I number/I of/O smaller/I ,/I non-communists/I parties/I ./O

Some tag sequences are inconsistent, such as the sequence O B. An automatic annotator can refuse such sequences, mapping them to a plausible annotation. That is, in the example above, to change the B tag into an I tag.

### 14.10.2 The IOB2 Tagset

From its original definition, Tjong Kim Sang and Veenstra (1999) proposed slight changes to the IOB format. The most widespread variant is called IOB2 or BIO, where the first word in a group receives the B tag (begin), and the following words the I tag. As for IOB, words outside the groups are annotated with the O tag. Using BIO, the two examples would be annotated as:

The/B government/I has/O other/B agencies/I and/I instruments/I for/O pursuing/O these/B other/I objectives/I ./O

Even/O Mao/B Tse-tung/I 's/B China/I began/O in/O 1949/B with/O a/B partnership/I between/O the/B communists/I and/O a/B number/I of/O smaller/B ,/I non-communists/I parties/I ./O

The BIO annotation scheme gained acceptance from the conferences on Computational Natural Language Learning (CoNLL 2000, see Sect. 14.14) that adopted it and went popular enough so that many people now use the term “IOB scheme” when they actually mean BIO.

### 14.10.3 The BIOES Tagset

The BIOES tagset is a third scheme, where B tag stands for beginning of group, I, for inside of group, E, for end of group, S, for group consisting of a single word, and finally O for outside. Using BIOES, the two examples are annotated as:

The/B government/E has/O other/B agencies/I and/I instruments/E for/O pursuing/O these/B other/I objectives/E ./O

Even/O Mao/B Tse-tung/E 's/B China/E began/O in/O 1949/S with/O a/B partnership/E between/O the/B communists/E and/O a/B number/E of/O smaller/B ,I non-communists/I parties/E ./O

### 14.10.4 Extending BIO to Two or More Groups

We can extend the BIO scheme to annotate two or more group categories. This is straightforward: We just need to use tags with a type suffix as for instance the tagset {B-Type1, I-Type1, B-Type2, I-Type2, O} to markup two different group types, Type1 and Type2.

CoNLL 2000 (Tjong Kim Sang & Buchholz 2000) again is an example such an annotation extension. The organizers used 11 different group types: noun phrases (NP), verb phrases (VP), prepositional phrases (PP), adverb phrases (ADVP), subordinated clause (SBAR), adjective phrases (ADJP), particles (PRT), conjunction phrases (CONJ), interjections (INTJ), list markers (LST), and unlike coordinated phrases (UPC).<sup>2</sup>

The noun phrases, verb phrases, and prepositional phrases making up more than 90% of all the groups in the CoNLL 2000 corpus.

### 14.10.5 Annotation Examples from CoNLL 2000, 2002, and 2003

As we saw, the CoNLL shared task in 2000 used the BIO (IOB2) tagset to annotate syntactic groups. Figure 14.12 shows an example of it with the sentence:

He reckons the current account deficit will narrow to only £1.8 billion in September.

whose annotation is:

[<sub>NG</sub> He <sub>NG</sub>] [<sub>VG</sub> reckons <sub>VG</sub>] [<sub>NG</sub> the current account deficit <sub>NG</sub>] [<sub>VG</sub> will narrow <sub>VG</sub>]  
[<sub>PG</sub> to <sub>PG</sub>] [<sub>NG</sub> only £1.8 billion <sub>NG</sub>] [<sub>PG</sub> in <sub>PG</sub>] [<sub>NG</sub> September <sub>NG</sub>].

---

<sup>2</sup> We feel that the word “phrase” has a misleading sense here. Most people in the field would understand it differently. The CoNLL 2000 phrases correspond to what we call group or chunk in this book: nonrecursive syntactic groups.

**Fig. 14.12** An annotation example of syntactic groups using the CoNLL 2000 extended BIO scheme. The parts of speech are predicted using Brill's tagger, while the groups are extracted from the Penn Treebank. The noun groups and verb groups are highlighted in *light blue* and *light green*, respectively. After data provided by Tjong Kim Sang and Buchholz (2000)

Words	POS	Groups
He	PRP	B-NP
reckons	VBZ	B-VP
the	DT	B-NP
current	JJ	I-NP
account	NN	I-NP
deficit	NN	I-NP
will	MD	B-VP
narrow	VB	I-VP
to	TO	B-PP
only	RB	B-NP
£	#	I-NP
1.8	CD	I-NP
billion	CD	I-NP
in	IN	B-PP
September	NNP	B-NP
.	.	O

and where the prepositional groups are limited to the preposition to avoid recursive groups. The dataset format is an older variant of CoNLL-U (Sect. 12.3) and has less information. It consists of only three columns:

1. The first column contains the sentence words with one word per line and a blank line after each sentence.
2. The second column contains the predicted parts of speech of the words. The CoNLL 2000 organizers used Brill's tagger (Brill 1995) trained on the Penn Treebank (Marcus et al. 1993) to assign these parts of speech (Tjong Kim Sang & Buchholz 2000). The POS tags are derived from the Penn Treebank and are now considered legacy. In multilingual corpora, they are replaced by the universal parts of speech, see Sect. 12.2;
3. The third column contains the groups with the manually assigned tags.

The topic of CoNLL 2002 and 2003 shared tasks was to annotate named entities. These tasks reused the ideas laid down in CoNLL 2001 with the BIO (IOB2) and IOB tag sets:

- The CoNLL 2002 annotation (Tjong Kim Sang 2002) consists two columns, the first one for the words and the second one for the named entities with four categories, persons (PER), organizations (ORG), locations (LOC), and miscellaneous (MISC). CoNLL 2002 uses BIO (IOB2). Figure 14.13, left part, shows the annotation of the sentence:

[*PER* Wolff *PER*], a journalist currently in [*LOC* Argentina *LOC*], played with [*PER* Del Bosque *PER*] in the final years of the seventies in [*ORG* Real Madrid *ORG*].

- The CoNLL 2003 annotation (Tjong Kim Sang and De Meulder 2003) has four columns: the words, parts of speech, syntactic groups, and named entities.

CoNLL 2002		CoNLL 2003			
Words	Named entities	Words	POS	Groups	Named entities
Wolff	B-PER	U.N.	NNP	I-NP	I-ORG
,	O	official	NN	I-NP	O
currently	O	Ekeus	NNP	I-NP	I-PER
a	O	heads	VBZ	I-VP	O
journalist	O	for	IN	I-PP	O
in	O	Baghdad	NNP	I-NP	I-LOC
Argentina	B-LOC	.	.	O	O
,	O				
played	O				
with	O				
Del	B-PER				
Bosque	I-PER				
in	O				
the	O				
final	O				
years	O				
of	O				
the	O				
seventies	O				
in	O				
Real	B-ORG				
Madrid	I-ORG				
.	O				

**Fig. 14.13** Annotation examples of named entities using the CoNLL 2002 (*left*) and CoNLL 2003 (*right*) IOB schemes. CoNLL 2002 has two columns and uses BIO (IOB2). CoNLL 2003 has four columns and uses IOB for the groups and named entities. After data provided by Tjong Kim Sang (2002) and Tjong Kim Sang and De Meulder (2003)

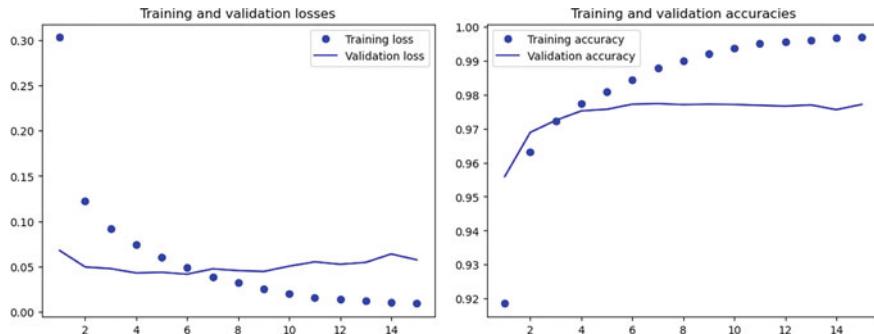
Both the syntactic groups and named entities use the original IOB scheme. Figure 14.13, right part, shows the annotation of the sentence:

[*ORG* U.N. *ORG*] official [*PER* Ekeus *PER*] heads for [*LOC* Baghdad *LOC*].

## 14.11 Recurrent Networks for Named Entity Recognition

In Sect. 14.8, we used recurrent networks in the annotation of a sequence of words with a sequence of parts of speech. We can apply them to recognize groups or named entities (NER). We keep the words as input, but we use either groups or named entities tags as output. The architecture is exactly the same otherwise.

To illustrate this, let us apply the LSTM architecture to NER tagging with the English dataset from CoNLL 2003. As with the previous experiments, we train the



**Fig. 14.14** LSTM training curves for named entity recognition. Loss and accuracy over the epochs

model on the training set, monitor the accuracy on the validation set, and evaluate the performance on the test set.

A simple architecture consists of an embedding layer, some dropout, a recurrent layer, and a linear layer to make the decision. As parameters, we used frozen GloVe 100 embeddings, two bidirectional LSTM layers, a dropout of 20%, a hidden size of 128, a batch size of 32, and nadam as optimizer with a learning rate of 0.001. We measure the performance with the CoNLL 2003 script that computes the F1 score for all the types of named entities and returns the macro-average.

Figure 14.14 shows the loss and accuracy over the epochs. We can see that the accuracy and loss reach an optimal value at epochs 6 or 7. We run again the model with 7 epochs and we obtain a CoNLL score of 85.30% on the test set.

## 14.12 Conditional Random Fields

A named entity tagger produces a sequence of tags, where a given tag obviously depends on the previous ones and sets constraints on the subsequent ones: For instance, an `End` tag, in the BIOES annotation, can only be preceded by an `Inside` or a `Begin`, and, conversely, an `Inside` tag can only be followed by an `Inside` or an `End`.

In Sect. 10.8, we modeled the word transitions probabilities with Bayes rules. In this section, we will estimate the tag probabilities using logistic regression instead. Such a technique is called **conditional random fields** (CRF) (Lafferty et al. 2001). Conditional random fields come with many variants. Here we will consider a simple form: the **linear chain**.

Denoting  $\mathbf{y}$  the output, here a sequence of tags, and  $\mathbf{x}$ , a sequence of inputs, consisting for instance of the words and the characters, we will try to maximize  $P(\mathbf{y}|\mathbf{x})$ , i.e.  $\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} P(\mathbf{y}|\mathbf{x})$ .

As we want the output sequence to depend on the input and on previously predicted output, we rewrite  $P(\mathbf{y}|\mathbf{x})$  as a joint probability,  $P(\mathbf{y}, \mathbf{x})$ , and, more precisely, as:

$$P(\mathbf{y}|\mathbf{x}) = \frac{P(\mathbf{y}, \mathbf{x})}{\sum_{\mathbf{y}' \in Y} P(\mathbf{y}', \mathbf{x})},$$

where  $\mathbf{y}'$  denotes a sequence,  $Y$ , the set of all the possible sequences, and  $\sum_{\mathbf{y}' \in Y} P(\mathbf{y}', \mathbf{x})$  is a normalizing factor to have a sum of probabilities of one (Sutton & McCallum 2011, p. 288).

To represent the features, conditional random fields use weighted **indicator functions**, also called feature functions. An indicator function is similar to the dummy variables or one-hot encoding we saw in Sect. 6.3. The function has either the value one, when a certain condition is met, or zero otherwise. For a sequence, it can be, for instance, a transition  $(y_{j-1}, y_j)$  at a certain position index  $j$  such as (`Begin`, `Inside`) at position 5 for a given sentence  $\mathbf{x}$ .

### 14.12.1 Modeling the $y$ Transitions

If we limit us to the possible transitions of  $y$  with the BIOES tagset (Sect. 14.10.3) and if  $\mathcal{Y}$  is the complete tagset, we will have  $|\mathcal{Y}|^2$  indicator functions. Table 14.8 shows the set of functions for a NER tagset without categories. There are five tags, and hence 25 indicator functions. For each pair of tags, Table 14.8 shows the function that will be equal to 1 for this transition. For instance, we have  $f_9(y_{j-1}, y_j) = 1$ , when we have the transition I → E and 0 for all the other transitions.

For one position  $j$ , the CRF probability is defined as the exponential of:

$$\sum_{i=1}^{|\mathcal{Y}|^2} w_i f_i(y_{j-1}, y_j),$$

**Table 14.8** Indicator functions fro the BIOES tagset

$y_{j-1} \setminus y_j$	B	I	O	E	S
B	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$
I	$f_6$	$f_7$	$f_8$	$f_9$	$f_{10}$
O	$f_{11}$	$f_{12}$	$f_{13}$	$f_{14}$	$f_{15}$
E	$f_{16}$	$f_{17}$	$f_{18}$	$f_{19}$	$f_{20}$
S	$f_{21}$	$f_{22}$	$f_{23}$	$f_{24}$	$f_{25}$

and, for the whole sequence of length  $N$ , as the product of all the probabilities:

$$\prod_{j=1}^N \exp \sum_{i=1}^{|\mathcal{Y}|^2} w_i f_i(y_{j-1}, y_j).$$

To have a sum of probabilities equals to 1, we need a normalizing factor, which is the sum over all the possible  $\mathbf{y}'$  sequences. We have then:

$$P(\mathbf{y}|\mathbf{x}) = \frac{\prod_{j=1}^N \exp \sum_{i=1}^{|\mathcal{Y}|^2} w_i f_i(y_{j-1}, y_j)}{\sum_{\mathbf{y}' \in Y} \prod_{j=1}^N \exp \sum_{i=1}^{|\mathcal{Y}|^2} w_i f_i(y'_{j-1}, y'_j)}.$$

and we find the  $w_i$  weights with a gradient descent.

### 14.12.2 Adding the $\mathbf{x}$ Input

In our previous presentation, we omitted the  $\mathbf{x}$  input. Of course, it plays a role in the prediction and that is why we need to include it in the probability. To take it into account, we rewrite the CRF probability for position  $j$  as:

$$\exp \sum_{i=1}^K w_i f_i(y_{j-1}, y_j, \mathbf{x}_j),$$

where  $\mathbf{x}_j$  is the input at index  $j$  and  $K$  is the number of indicator functions.  $\mathbf{x}_j$  can simply be the word at index  $j$  or a window of words around  $x_j$ . An indicator function can be the triple  $(y_{i-1}, y_j, x_i)$ , for instance the tags `Outside` and `Begin` and the word `the`, or more simply, the pairs  $(y_{i-1}, y_j)$  and  $(y_j, x_i)$ .

For the whole sequence, we have:

$$\prod_{j=1}^N \exp \sum_{i=1}^K w_i f_i(y_{j-1}, y_j, \mathbf{x}_j),$$

that we normalize as above:

$$P(\mathbf{y}|\mathbf{x}) = \frac{\prod_{j=1}^N \exp \sum_{i=1}^K w_i f_i(y_{j-1}, y_j, \mathbf{x}_j)}{\sum_{\mathbf{y}' \in Y} \prod_{j=1}^N \exp \sum_{i=1}^K w_i f_i(y'_{j-1}, y'_j, \mathbf{x}_j)}.$$

### 14.12.3 Predicting a Sequence

In the prediction step, the optimal sequence is given by:

$$\arg \max_{\mathbf{y}} \sum_{j=1}^N \sum_{i=1}^K w_i f_i(y_{j-1}, y_j, \mathbf{x}_j),$$

where we set aside the denominator as it is the same for all the sequences and the exponential.

However, we do not have access to  $\mathbf{y}$  in advance. We could use a brute-force method: Generate all the possible sequences and keep the highest probability, and hence the best label sequence. This would not scale of course and we need to resort to a Viterbi search instead, as in Sect. 13.4.3.

### 14.12.4 Adding a CRF Layer to a LSTM Network

When dealing with a practical application, the potential number of indicator functions can be very large making their selection quite tricky. To solve this, we will follow Lample et al. (2016) and their implementation of a sequence tagger for named entity recognition. These authors proposed a pipeline architecture, where they added a CRF layer to a bidirectional LSTM. In this model,

1. The input consists of a sequence of word embeddings or word embeddings and characters. The authors used the skipgram algorithm to derive the word embeddings, see Sect. 11.9.4;
2. The embeddings are passed to a bidirectional LSTM layer;
3. The LSTM output is passed to a linear layer and results in a sequence of tag predictions. At index  $i$  of the sequence, each possible tag is associated with a logit. We could relate these logits to probabilities. We just need to apply them a softmax function;
4. The predictions are finally passed to a CRF layer. This layer uses a set of indicator functions limited to bigram output transitions and to pairs of LSTM output and output.

Given a word sequence as input:

$$\mathbf{x} = (x_1, x_2, \dots, x_N),$$

Lample et al. (2016) defined the score of the predicted tag sequence:

$$\mathbf{y} = (y_1, y_2, \dots, y_N)$$

as

$$\text{Score}(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^N a_{y_i, y_{i+1}} + \sum_{i=1}^N P_{i, y_i},$$

where  $a_{i,j}$  is the transition score from tag  $i$  to tag  $j$  and  $P_{i,y_i}$  is the logit of tag  $y_i$  at index  $i$  in the sequence obtained from the LSTM. This latter term is called the emission score.

The conditional probability is:

$$P(\mathbf{y}|\mathbf{x}) = \frac{\exp \text{Score}(\mathbf{x}, \mathbf{y})}{\sum_{\mathbf{y}' \in Y} \exp \text{Score}(\mathbf{x}, \mathbf{y}')}$$

or using logarithms:

$$\log P(\mathbf{y}|\mathbf{x}) = \text{Score}(\mathbf{x}, \mathbf{y}) - \log \sum_{\mathbf{y}' \in Y} \exp \text{Score}(\mathbf{x}, \mathbf{y}').$$

To fit a CRF model, we need an annotated corpus from which we extract the  $X$  and  $Y$  matrices. As for logistic regression, we maximize the log-likelihood of the observed sequences (or minimize the negative log-likelihood). We find the optimal CRF and LSTM weights with a gradient descent.

PyTorch has no built-in module for CRF as of today. However, the PyTorch documentation provides a good tutorial on how to implement one for named entity recognition<sup>3</sup> (Guthrie 2023). The example program uses a notation that is close to that in Lample et al. (2016).

Adding a CRF layer to the LSTM, Lample et al. (2016) report a score of 90.20 on the English part of the CoNLL 2003 dataset and even 90.94 when including the characters in the input.

## 14.13 Tokenization

Another application of sequence annotation on a completely different topic is tokenization. We already saw in Sect. 9.2.4 that we could use classifiers to tokenize a text. Here we will describe a technique proposed by Shao et al. (2018) that uses recurrent networks.

The dataset consists of pairs of sequences, where as input we have the raw characters of a sentence and as output, the tags marking the words and their boundaries. Shao et al. (2018) aligned the characters with the BIOES tags from

---

<sup>3</sup> [https://pytorch.org/tutorials/beginner/nlp/advanced\\_tutorial.html](https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html).

Sect. 14.10.3, where **B** is the beginning of a token, **E**, its end, **I** possible characters inside a token, and **S**, a single-character token. They also used the **X** tag to mark the delimiters, mostly spaces as in this example abridged from their paper:

Char.	On considère qu'environ 50 000 Allemands du Wartheland ont péri.
Tags	BEXBIIIIIIXBIEBIIIIEXBIIIIEXBIIIIIIEXBÉXBIIIIIIEXBIEXBIIIES

Shao et al. (2018) used the corpora from the Universal Dependencies to build the training set. They created the input and output sequences from unprocessed sentences and their tokenization in the CoNLL-U format (Sect. 12.3). The model consists of:

1. An embedding layer, where the embedding represents the current character. For Asian languages, the embedding is the concatenation of the current character, the bigram of the current character and the preceding one, and the trigram with the two surrounding characters are concatenated. We can relate this to what we saw in Sect. 11.10, here with recurrent networks;
2. Bidirectional gated recurrent units (GRU), a simpler variant of the LSTM architecture, and
3. A CRF layer.

Once trained, the application of the model to a sequence of characters enables us to extract the tokens from the tags: These are simply all the matches of the **BI\*E** and **S** patterns in the output string. For the example above, this corresponds to *On*, *considère*, *qu'*, *environ*, *50 000*, *Allemands*, *du*, *Wartheland*, *ont*, *péri*, and the period **.** that ends the sentence. Note that a white space tokenizer would have wrongly tokenized the multiword *qu'environ* as well as the number *50 000*.

The **B̄** and **É** tags are variants of **B** and **E**. In the sentence, they mean that the word *du* should be further transduced into *de* and *le* as in Sect. 14.3 and Table 14.3. As these transductions are overall unambiguous in French, the authors used a dictionary to process them.

Shao et al. (2018) reported the best tokenization performance when their paper was published. Their model can also be extended to sentence segmentation.

## 14.14 Further Reading

Part-of-speech tagging has a long history in language processing. Brill's program marked a breakthrough when he applied symbolic techniques with rules induced from annotated corpora. Roche and Schabes (1995) proposed a dramatic optimization of his tagger using transducers.

Linear classifiers that we saw in this chapter form another line of efficient tools. Ratnaparkhi (1996) is an early example with logistic regression. In addition to the local classification, Ratnaparkhi optimized the complete part-of-speech sequence.

He multiplied the output probabilities from each tagging operation and searched the tag sequence so that the product:

$$\hat{T} = \arg \max_{t_1, t_2, t_3, \dots, t_n} \prod_{i=1}^n P(t_i | w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}, t_{i-2}, t_{i-1}, t_{i-1})$$

reaches a maximum. Ratnaparkhi applied a beam search to find the optimal sequence.

The development of group detection and named entity recognition has been mainly driven by applications without concern for a specific linguistic framework. This was a notable difference from many other areas of language processing at that time. Due to the simplicity of the methods involved, partial or shallow parsing attracted considerable interest in the 1990s.

Ramshaw and Marcus (1995) created the IOB tagset and they adapted Brill's (1995) algorithm to learn rules to detect groups from annotated corpora. Kudoh and Matsumoto (2000) applied classifiers based on support vector machines for group detection. Conditional random fields are often used as a last layer of recurrent neural networks. Sutton and McCallum (2011) provide a comprehensive description of them with the theory, algorithms, and a few examples.

Partial parsing was the topic of a series of conferences on Computational Natural Language Learning (CoNLL). Each year, the CoNLL conference organizes a “shared task” where it provides an annotated training set. Participants can train their system on this set, evaluate it on a common test set, and report a description of their algorithms and results in proceedings. In 1999, the shared task was dedicated to noun group chunking;<sup>4</sup> in 2000 it was extended to other chunks;<sup>5</sup> in 2001 the topic was the identification of clauses;<sup>6</sup> and in 2002 and 2003 the task was multilingual named entity recognition.<sup>7,8</sup> The CoNLL sites and proceedings are extremely valuable as they provide datasets, annotation schemes, a good background literature, and an excellent idea of the state of the art.

---

<sup>4</sup> <https://www.cnts.ua.ac.be/conll199/npb/>.

<sup>5</sup> <https://www.cnts.ua.ac.be/conll2000/chunking/>.

<sup>6</sup> <https://www.cnts.ua.ac.be/conll2001/clauses/>.

<sup>7</sup> <https://www.cnts.ua.ac.be/conll2002/ner/>.

<sup>8</sup> <https://www.cnts.ua.ac.be/conll2003/ner/>.

# Chapter 15

## Self-Attention and Transformers



After feedforward and recurrent networks in Chaps. 8, *Neural Networks* and 14, *Part-of-Speech and Sequence Annotation*, transformers (Vaswani et al. 2017) are a third form of networks (a kind of feedforward in fact). This architecture, based on the concept of attention, features two twin processing pipelines called the *encoder* and the *decoder*.

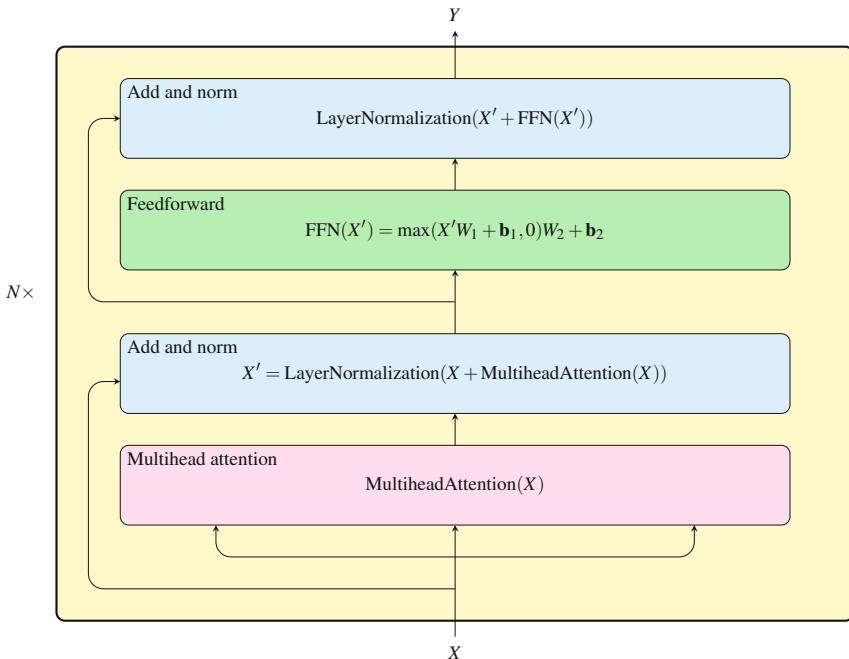
### 15.1 The Transformer Architecture

Given an input sequence of length  $n$  consisting of embeddings of dimensionality  $d_{\text{model}}$ , the first part of a transformer, the encoder, outputs a sequence of embeddings of identical length with the same dimensionality. The input embeddings usually represent words, subwords, or characters and, adding a classifier to the output, the encoder produces a sequence of tags of equal length. This application corresponds to a sequence annotation as in Chap. 14, *Part-of-Speech and Sequence Annotation* and Fig. 14.3.

More precisely, an encoder consists of a stack of  $N$  identical layers, where each layer is a pipeline of two main components shown in Fig. 15.1:

1. An attention mechanism that transforms the  $X$  input vectors into context-dependent vectors; This attention is applied in parallel and the results, called heads, are concatenated yielding a multihead attention, see the pink block in Fig. 15.1;
2. The multihead attention output is then passed to a feed-forward network to produce the final output,  $Y$ , see the green block in Fig. 15.1. The  $X$  and  $Y$  matrices have identical size.

In this chapter, we will start with the description of attention and the encoder part of transformers and we will outline their implementation with programs in Python.



**Fig. 15.1** The encoder layer stacked  $N$  times. The  $X$  and  $Y$  matrices have identical size. After Vaswani et al. (2017)

PyTorch has also built-in modules for both attention and encoder layers that are ready to use. We will describe them as they make programming much easier.

We can use both the encoder and the decoder as standalone components. In the next chapters, we will describe how to train the encoder transformer and then we will see the complete encoder-decoder architecture.

## 15.2 Self-Attention

Once trained, the values of the word embeddings we have seen in Chap. 11, *Dense Vector Representations*, either Word2Vec or GloVe, are immutable whatever the words that surround them. In addition, these embeddings ignore the possible senses of a word. For instance, *note*, while having at least two senses, as in *musical note* and *taking note*, has only one single embedding vector. Such embeddings are said to be static or noncontextual. While static embeddings improve the representation of words over one-hot encodings, their insensitivity to context also entails an obvious limitation.

Self-attention is a mechanism that will enable us to change the embedding vectors depending on the context. We will show how this works with this quote from the *Odyssey*, book I:

I must go back to my ship and to my crew.

and compare the embeddings of *ship* with those of the same word in the sentence:

We process and ship your order [in the most cost-efficient way possible]

excerpted from an Amazon.com commercial page.<sup>1</sup>

### 15.2.1 GloVe Static Embeddings

As static embeddings, we will use the GloVe 50-dimensional vectors (Pennington et al. 2014) in this chapter.<sup>2</sup> We read the GloVe embeddings from the glove.6B.50d.txt file with the `read_embeddings()` function from Sect. 14.7 that stores the words and embeddings in a dictionary.

```
embedding_file = '/PATH/glove.6B.50d.txt'
embeddings_dict = read_embeddings(embedding_file)
```

We create two lists of words from our sentences:

```
sentence_odyssey = 'I must go back to my ship and to my crew'
sentence_amazon = 'We process and ship your order'
```

```
words_o = sentence_odyssey.lower().split()
words_a = sentence_amazon.lower().split()
```

and a PyTorch tensor,  $X$ , from the embedding dictionary for *Odyssey*'s quote:

```
def embedding_matrix(words, embeddings_dict):
    X = [embeddings_dict[word] for word in words]
    X = torch.stack(X)
    return X

X = embedding_matrix(words_o, embeddings_dict)
```

The *ship* embedding is a vector of 50 coordinates starting with:

$$\mathbf{emb}(\textit{ship}) = (1.5213, 0.1052, 0.3816, -0.5080, 0.0324, -0.1348, -1.2474, \dots)$$

---

<sup>1</sup> <https://www.amazon.com/gp/help/customer/display.html?nodeId=GV8H5D3MMAR7JBLF>.

<sup>2</sup> <https://nlp.stanford.edu/projects/glove/>.

### 15.2.2 Using Weighted Cosines to Create Contextual Embeddings

We have seen in Sect. 11.6 that we can evaluate the semantic similarity between two words,  $w_i$  and  $w_j$ , with the cosine of their embeddings,  $\text{emb}(w_i)$  and  $\text{emb}(w_j)$ , defined as:

$$\cos(\text{emb}(w_i), \text{emb}(w_j)) = \frac{\text{emb}(w_i) \cdot \text{emb}(w_j)}{\|\text{emb}(w_i)\| \cdot \|\text{emb}(w_j)\|}.$$

With self-attention, we will compute a similarity between all the pairs of words in a sentence or a sequence. Using this similarity, we will be able to create new embeddings modeling their mutual influence.

Practically, we compute the cosines of the pairs with the program in Sect. 5.6.3. Figure 15.2 shows these cosines for all the pairs of embeddings for Homer's quote. For a given word, the cosines of the surrounding words reflect how much meaning they share with it. For instance, we have a score 0.78 for the pair *ship* and *crew*, two arguably related words.

We can use the cosines as weights and define contextual embeddings as the weighted sums of the initial embeddings. Using the values in Fig. 15.2, yellow row, the contextual embedding of *ship* in this sentence is then:

$$\begin{aligned} \text{ctx\_emb}(\textit{ship}) = & 0.35 \cdot \text{emb}(i) + 0.42 \cdot \text{emb}(\textit{must}) + 0.41 \cdot \text{emb}(\textit{go}) + \\ & 0.49 \cdot \text{emb}(\textit{back}) + 0.54 \cdot \text{emb}(\textit{to}) + 0.38 \cdot \text{emb}(\textit{my}) + \\ & 1.00 \cdot \text{emb}(\textit{ship}) + 0.46 \cdot \text{emb}(\textit{and}) + 0.54 \cdot \text{emb}(\textit{to}) + \\ & 0.38 \cdot \text{emb}(\textit{my}) + 0.78 \cdot \text{emb}(\textit{crew}), \end{aligned}$$

where *ship* will receive 78% of the *crew* embedding.

	i	must	go	back	to	my	ship	and	to	my	crew
i	1.00	0.75	0.86	0.76	0.73	0.90	0.35	0.65	0.73	0.90	0.42
must	0.75	1.00	0.85	0.68	0.87	0.69	0.42	0.69	0.87	0.69	0.45
go	0.86	0.85	1.00	0.84	0.84	0.81	0.41	0.68	0.84	0.81	0.49
back	0.76	0.68	0.84	1.00	0.83	0.76	0.49	0.77	0.83	0.76	0.51
to	0.73	0.87	0.84	0.83	1.00	0.68	0.54	0.86	1.00	0.68	0.51
my	0.90	0.69	0.81	0.76	0.68	1.00	0.38	0.63	0.68	1.00	0.44
ship	<b>0.35</b>	<b>0.42</b>	<b>0.41</b>	<b>0.49</b>	<b>0.54</b>	<b>0.38</b>	<b>1.00</b>	<b>0.46</b>	<b>0.54</b>	<b>0.38</b>	<b>0.78</b>
and	0.65	0.69	0.68	0.77	0.86	0.63	0.46	1.00	0.86	0.63	0.49
to	0.73	0.87	0.84	0.83	1.00	0.68	0.54	0.86	1.00	0.68	0.51
my	0.90	0.69	0.81	0.76	0.68	1.00	0.38	0.63	0.68	1.00	0.44
crew	0.42	0.45	0.49	0.51	0.51	0.44	<b>0.78</b>	0.49	0.51	0.44	<b>1.00</b>

**Fig. 15.2** Cartesian product of the cosines of the pairs of embeddings with *ship* and *crew* in bold. This table uses GloVe 50d vectors trained on Wikipedia 2014 and Gigaword 5

**Fig. 15.3** Cartesian product of the cosines of the pairs of embeddings with *ship* and *order* in bold

	we	process	and	ship	your	order
we	1.00	0.64	0.70	0.36	0.75	0.64
process	0.64	1.00	0.61	0.29	0.52	0.67
and	0.70	0.61	1.00	0.46	0.58	0.69
ship	0.36	0.29	0.46	<b>1.00</b>	0.37	<b>0.52</b>
your	0.75	0.52	0.58	0.37	1.00	0.63
order	0.64	0.67	0.69	<b>0.52</b>	0.63	<b>1.00</b>

We compute these weighted sums for all the words, and hence the new embeddings, with a simple matrix product. Denoting  $C$  the cosine matrix in Fig. 15.2 and the  $X$  the matrix of initial embeddings arranged in rows, the contextual embeddings are the rows of  $CX$ . For *ship*, we have the 50-dimensional vector starting with:

$$\begin{aligned} \text{ctx\_emb}(ship) \\ = (3.2319, 0.6408, 1.4718, -2.3434, 2.2358, -0.4188, -4.1002, \dots) \end{aligned}$$

Comparing the sentence from the *Odyssey* with *We process and ship your order* in Fig. 15.3, we see that, this time, the *ship* embedding receives 52% from the *order* embedding reflecting thus a completely different context.

### 15.2.3 Dot-Product Attention

The idea of self-attention proposed by Vaswani et al. (2017) is very similar to that of the cosines; it simply replaces them with a scaled dot-product. As in the previous section, we denote  $X$  the embeddings of the sentence words and  $\lambda$  the scaling factor. The matrix of scaled dot-products for all the pairs is the product:

$$\lambda XX^\top.$$

This matrix is equivalent to the weights in Fig. 15.2 and has the same size. As scaling factor, Vaswani et al. (2017) use  $\frac{1}{\sqrt{d_{\text{model}}}}$ , where  $d_{\text{model}}$  is the dimensionality of the input embeddings. As we use GloVe 50d in our examples, we have  $d_{\text{model}} = 50$ . We normalize these weights with the softmax function:

$$\text{softmax} \left( \frac{XX^\top}{\sqrt{d_{\text{model}}}} \right),$$

	i	must	go	back	to	my	ship	and	to	my	crew
i	0.36	0.05	0.07	0.05	0.04	0.19	0.01	0.02	0.04	0.19	0.01
must	0.14	0.20	0.10	0.06	0.11	0.10	0.03	0.05	0.11	0.10	0.02
go	0.18	0.09	0.14	0.09	0.08	0.13	0.02	0.04	0.08	0.13	0.02
back	0.14	0.05	0.09	0.19	0.08	0.12	0.03	0.06	0.08	0.12	0.03
to	0.11	0.11	0.09	0.09	0.15	0.08	0.04	0.07	0.15	0.08	0.03
my	0.19	0.03	0.05	0.04	0.03	0.29	0.01	0.02	0.03	0.29	0.01
ship	0.03	0.03	0.03	0.04	0.05	0.03	<b>0.55</b>	0.03	0.05	0.03	<b>0.13</b>
and	0.10	0.08	0.07	0.10	0.12	0.09	0.04	0.15	0.12	0.09	0.04
to	0.11	0.11	0.09	0.09	0.15	0.08	0.04	0.07	0.15	0.08	0.03
my	0.19	0.03	0.05	0.04	0.03	0.29	0.01	0.02	0.03	0.29	0.01
crew	0.06	0.05	0.05	0.06	0.05	0.06	0.21	0.04	0.05	0.06	0.31

**Fig. 15.4** Self-attention weights. This table uses GloVe 50d vectors trained on Wikipedia 2014 and Gigaword 5

defined as (see Sect. 8.6.2):

$$\text{softmax}(x_1, x_2, \dots, x_j, \dots, x_k) = \left( \frac{e^{x_1}}{\sum_{i=1}^k e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^k e^{x_i}}, \dots, \frac{e^{x_j}}{\sum_{i=1}^k e^{x_i}}, \dots, \frac{e^{x_k}}{\sum_{i=1}^k e^{x_i}} \right)$$

so that each row has a sum of one.

Figure 15.4 shows the attention weights obtained with the dot-product and the same GloVe input embeddings as those used with the cosines in Fig. 15.2. The contextual embeddings are the weighted sums of the initial ones. For *ship*, we have:

$$\begin{aligned} \text{ctx\_emb}(ship) &= 0.03 \cdot \text{emb}(i) + 0.03 \cdot \text{emb}(must) + 0.03 \cdot \text{emb}(go) + \\ &\quad 0.04 \cdot \text{emb}(back) + 0.05 \cdot \text{emb}(to) + 0.03 \cdot \text{emb}(my) + \\ &\quad 0.55 \cdot \text{emb}(ship) + 0.03 \cdot \text{emb}(and) + 0.05 \cdot \text{emb}(to) + \\ &\quad 0.03 \cdot \text{emb}(my) + 0.13 \cdot \text{emb}(crew), \end{aligned}$$

where this word keeps 55% of its initial value and gets 13% from *crew*.

As with the cosines, we compute the contextual embeddings of all the words in the sentence with the product of the weights by  $X$ :

$$\text{softmax} \left( \frac{XX^\top}{\sqrt{d_{\text{model}}}} \right) X.$$

### 15.2.4 Projecting the Input

The contextual embeddings result from the product of the weights by the input  $X$ . To keep track of their different contributions, self-attention does not use the original embeddings directly. It first multiplies  $X$  by three trainable matrices:  $W^Q$ ,  $W^K$ , and  $W^V$ . The size of these matrices is:

- For  $X$ , the number of input tokens by  $d_{\text{model}}$ ;
- For  $W^Q$  and  $W^K$ ,  $d_{\text{model}} \times d_k$ ; and
- For  $W^V$ ,  $d_{\text{model}} \times d_v$ .

In the rest of this chapter, we will use  $d_k = d_v$ .

For a row input vector,  $\mathbf{x}_i$  representing the token at index  $i$ , we have:

$$\begin{aligned}\mathbf{q}_i &= \mathbf{x}_i W^Q, \\ \mathbf{k}_i &= \mathbf{x}_i W^K, \\ \mathbf{v}_i &= \mathbf{x}_i W^V,\end{aligned}$$

where  $\mathbf{x}_i$  is of size  $1 \times d_{\text{model}}$  and  $\mathbf{q}_i$ ,  $\mathbf{k}_i$ , and  $\mathbf{v}_i$  are of size  $1 \times d_k$ . In the original implementation, Vaswani et al. (2017) used the value  $d_{\text{model}} = 512$ . For one single such attention module, they proposed  $d_k = d_{\text{model}}$ .

Using the complete input sequence (all the tokens),  $X$ , we have:

$$\begin{aligned}Q &= X W^Q, \\ K &= X W^K, \\ V &= X W^V,\end{aligned}$$

where  $Q$  is called the query,  $K$ , the key, and  $V$ , the value.

Following Vaswani et al. (2017), the final equation of self-attention is:

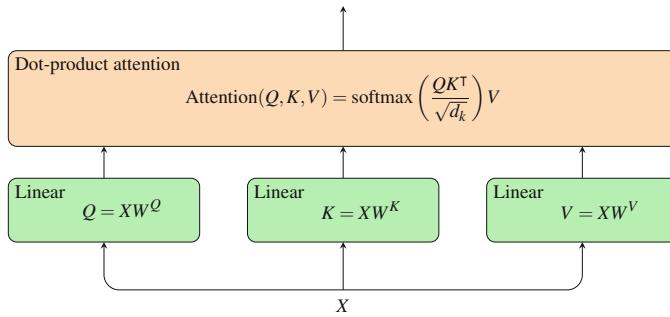
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where  $d_k$  is the number of columns of the matrices and  $\sqrt{d_k}$ , the scaling factor. Figure 15.5 shows a summary of self-attention.

### 15.2.5 Programming the Attention Function

We will now write the code to compute a self-attention with PyTorch. We first import the modules:

```
import math
import torch
import torch.nn.functional as F
```



**Fig. 15.5** Dot-product self-attention. After Vaswani et al. (2017)

The `attention` function is straightforward from the definition. It returns the attention weights and the contextual embeddings:

```
def attention(Q, K, V):
    d_k = K.size(dim=-1)
    attn_weights = F.softmax(Q @ K.T/math.sqrt(d_k), dim=-1)
    attn_output = attn_weights @ V
    return attn_output, attn_weights
```

We call the `attention()` function with a simplified equation, where we use identity matrices for  $W^Q$ ,  $W^K$ , and  $W^V$  i.e.  $XW^Q = XW^K = XW^V = X$ :

```
attn_output, attn_weights = attention(X, X, X)
```

As results, we obtain the attention weights shown in Fig. 15.4 and new contextual embeddings as for *ship*:

```
ctx_emb(ship)
= (1.0387, 0.1033, 0.3426, -0.4320, 0.2237, -0.0958, -0.9926, ...)
```

The size of the contextual embeddings matrix is the sequence length by the dimensionality of the embeddings,  $n \times d_{\text{model}}$ :

```
>>> attn_output.size()
torch.Size([11, 50])
```

here 11 words and 50-dimensional embeddings and  $n \times n$  for the weight matrix:

```
>>> attn_weights.size()
torch.Size([11, 11])
```

### 15.2.6 Adding the Query, Key, and Value Matrices

Once we have seen the effect of attention on a simple example, we can add the  $W^Q$ ,  $W^K$ , and  $W^V$  matrices to the computation. We encapsulate them in a class:

```
class Attention(nn.Module):
    def __init__(self, d_model, d_k):
        super().__init__()
        self.WQ = nn.Linear(d_model, d_k)
        self.WK = nn.Linear(d_model, d_k)
        self.WV = nn.Linear(d_model, d_k)

    def forward(self, X):
        attn_output, attn_weights = attention(self.WQ(X),
                                              self.WK(X),
                                              self.WV(X))
        return attn_output, attn_weights
```

We compute `d_model` from the size of the embeddings, here the values of the dictionary. For this, we read the first key and its value:

```
d_model = embeddings_dict[
    next(iter(embeddings_dict))].size(dim=-1)
```

and we obtain `d_model = 50`.

We create an attention module and apply it to the sentence embeddings with the statements:

```
>>> attn = Attention(d_model, d_model)
>>> attn_output, attn_weights = attn(X)
```

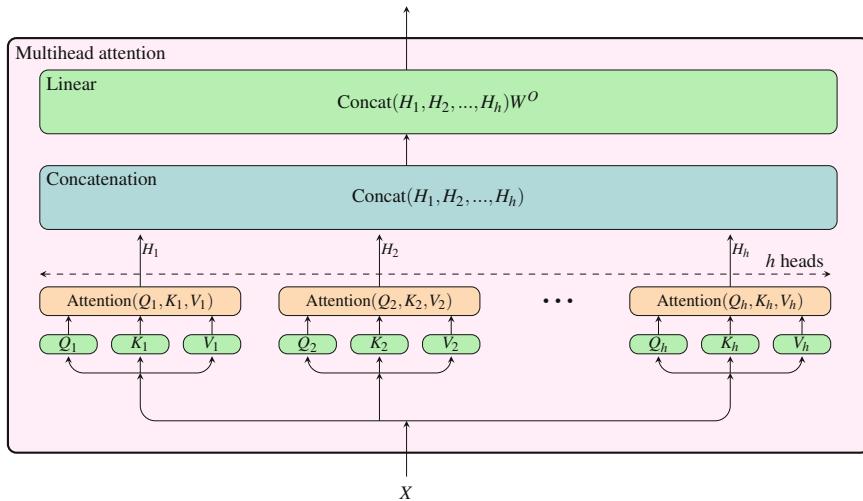
As with the `attention()` function, this yields contextual embeddings and a weight matrix. Both matrices have the same size as in the previous section, but their values are different as the linear modules have a random initialization.

## 15.3 Multihead Attention

Vaswani et al. (2017) found they could capture more information when they duplicated the attention in Fig. 15.5 into parallel modules. This is the idea of multihead attention that concatenates the results of the single attention modules and passes them to a linear function shown in Fig. 15.6.

To implement multihead attention with  $h$  heads, we create  $h$  triples of matrices  $(W_i^Q, W_i^K, W_i^V)$  with  $i$  ranging from 1 to  $h$ , where the matrices have the size  $d_{\text{model}} \times d_k$ . For each head, we compute an attention value:

$$H_i = \text{softmax} \left( \frac{Q_i K_i^\top}{\sqrt{d_k}} \right) V_i.$$



**Fig. 15.6** Multihead attention. After Vaswani et al. (2017)

We concatenate these heads and we apply the linear function in the form of matrix  $W^O$  to reduce the output dimension to  $d_{\text{model}}$ :

$$\text{Concat}(H_1, H_2, \dots, H_h)W^O.$$

The size of  $\text{Concat}(H_1, H_2, \dots, H_h)$  is the number of input tokens by  $h \cdot d_k$  and the size of  $W^O$  is  $h \cdot d_k \times d_{\text{model}}$ . In their implementation, Vaswani et al. (2017) used  $h = 8$ ,  $d_{\text{model}} = 512$ , and  $d_k = d_{\text{model}}/h = 64$  so that the multihead output dimension is the same that of a simple attention module. The size of  $W^O$  is then  $d_{\text{model}} \times d_{\text{model}}$ .

Using our `Attention` class, we can easily create a multihead attention. As input, we give the model dimension,  $d_{\text{model}}$ , and the number of heads. This number must be a divisor of  $d_{\text{model}}$ . We instantiate as many attention modules as we have heads and we store them in a Python list. We register the Python list as a list of modules to declare the matrix values as parameters.

In the forward pass, we apply the attention modules to the input to get the heads and their respective weights. We concatenate the heads and we apply them the output matrix  $W^O$ . We also return the average of the weight matrices:

```
class MultiheadAttention(nn.Module):
    def __init__(self, d_model, nhead):
        super().__init__()
        self.nhead = nhead
        d_k = d_model // nhead
        self.attn_modules = nn.ModuleList(
            [Attention(d_model, d_k)
             for i in range(nhead)])
        self.W0 = nn.Linear(d_model, d_model)
```

```

def forward(self, X):
    attn_heads, attn_weights = zip(
        *[attn_module(X)
          for attn_module in self.attn_modules])
    attn_output = self.W0(torch.cat(attn_heads, dim=-1))
    attn_weights = torch.sum(torch.stack(attn_weights),
                           dim=0)/self.nhead
    return attn_output, attn_weights

```

We create a multihead attention with a number of heads, 5, that is a divisor of  $d_{\text{model}}$ , here 50.

```

>>> nhead = 5
>>> multihead_attn = MultiheadAttention(d_model, nhead)

```

And we compute the contextual embeddings as in the previous examples:

```
>>> attn_output, attn_weights = multihead_attn(X)
```

We have now written a toy implementation of multihead attention to give an outline of its structure. We must be aware however that it is not optimized for speed and it cannot handle batches of input. Fortunately, PyTorch has a ready-to-use function that we will describe in Sect. 15.12

## 15.4 Residual Connections

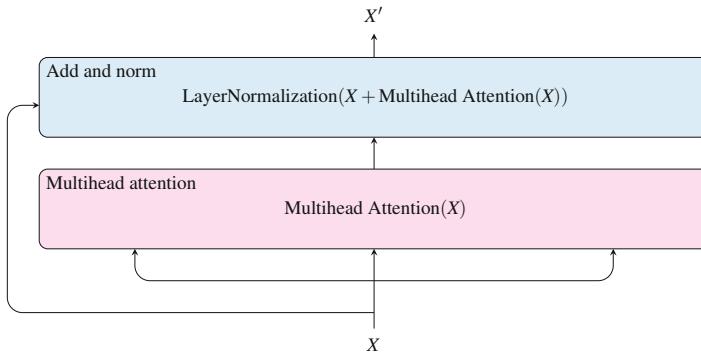
When optimizing a neural network, the convergence is often unstable. He et al. (2016) showed that it could be improved with a residual connection, where the input of a network is added to its output. When applied to the multihead attention module, the new output is:

$$X + \text{MultiheadAttention}(X).$$

This result is then passed to a layer normalization (Ba et al. 2016), where for each token at index  $i$  in the sequence, its embedding vector,  $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,d_{\text{model}}})$ , is normalized individually. This function is defined as a vector standardization followed by a Hadamard product with a gain and the addition of a bias. We have:

$$\begin{aligned} & \text{LayerNorm}(x_{i,1}, \dots, x_{i,j}, \dots, x_{i,d_{\text{model}}}) \\ &= \mathbf{g} \odot \left( \frac{x_{i,1} - \bar{x}_{i,\cdot}}{\sigma_{x_{i,\cdot}}}, \dots, \frac{x_{i,j} - \bar{x}_{i,\cdot}}{\sigma_{x_{i,\cdot}}}, \dots, \frac{x_{i,d_{\text{model}}} - \bar{x}_{i,\cdot}}{\sigma_{x_{i,\cdot}}} \right) \\ &+ \mathbf{b}, \end{aligned}$$

where  $j$  is a coordinate index,  $\bar{x}_{i,\cdot}$  and  $\sigma_{x_{i,\cdot}}$ , the mean and standard deviation of the embedding vector at index  $i$ .  $\mathbf{g}$  and  $\mathbf{b}$  are learnable parameter vectors of  $d_{\text{model}}$  dimensionality, initialized to 1 and 0, respectively.



**Fig. 15.7** Residual connection with layer normalization. After Vaswani et al. (2017)

The complete function is defined as (Fig. 15.7):

$$X' = \text{LayerNorm}(X + \text{MultiheadAttention}(X)).$$

We implement this residual network with the `nn.LayerNorm()` class from PyTorch. `nn.LayerNorm()` applies the normalization to the samples of a mini-batch. Here we use it for just one sequence.

```
class LayerNormAttention(nn.Module):
    def __init__(self, d_model, nhead):
        super().__init__()
        self.multihead_attn = MultiheadAttention(d_model,
                                                nhead)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, X):
        Y, _ = self.multihead_attn(X)
        return self.layer_norm(X + Y)
```

We create a layer-normalized attention object with:

```
>>> ln_m_attn = LayerNormAttention(d_model, nhead)
```

and we call it with:

```
>>> ln_m_attn(X)
```

As in the previous examples, this computes contextual embeddings of size  $n \times d_{\text{model}}$ .

## 15.5 Feedforward Sublayer

The last part of our encoder layer consists of a feedforward network with two linear modules and a ReLU function,  $\max(x, 0)$ , in-between:

$$\text{FFN}(X') = \max(X'W_1 + \mathbf{b}_1, 0)W_2 + \mathbf{b}_2$$

As with attention, we apply a layer normalization:

$$\text{LayerNormalization}(X' + \text{FFN}(X')).$$

The size of  $W_1$  is  $d_{\text{model}} \times d_{ff}$  and that is  $W_2$  is  $d_{ff} \times d_{\text{model}}$ .

## 15.6 The Encoder Layers

The encoder in Fig. 15.1 consists of a stack of  $N$  identical layers. Each layer is composed of a multihead attention and a feedforward network. The implementation of a layer is straightforward from its description. We just add the feedforward network in the previous class:

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nhead, d_ff=2048):
        super().__init__()
        self.multihead_attn = MultiheadAttention(d_model,
                                                nhead)
        self.layer_norm_1 = nn.LayerNorm(d_model)
        self.W1 = nn.Linear(d_model, d_ff)
        self.relu = nn.ReLU()
        self.W2 = nn.Linear(d_ff, d_model)
        self.layer_norm_2 = nn.LayerNorm(d_model)

    def forward(self, X):
        Xprime, _ = self.multihead_attn(X)
        Xprime = self.layer_norm_1(X + Xprime)

        Y = self.W2(self.relu(self.W1(Xprime)))
        Y = self.layer_norm_2(Xprime + Y)
        return Y
```

Using this class, we create an encoder layer for our GloVe embeddings with 5 heads:

```
>>> encoder_layer = TransformerEncoderLayer(d_model, nhead)
```

And we apply it to our input with:

```
>>> enc_layer_output = encoder_layer(X)
```

to get contextual embeddings from the layer.

## 15.7 The Complete Encoder

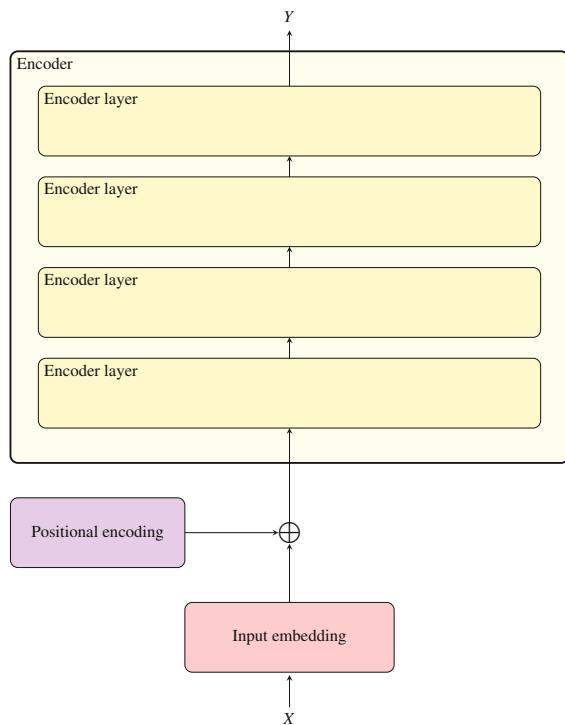
The complete encoder in Figs. 15.1 and 15.8 consists of a stack of  $N$  encoder layers. We have now programmed all the modules we need to build it. Let us call it `TransformerEncoder`.

The `TransformerEncoderLayer` class has one layer. We first create a layer object, `encoder_layer`, and we pass it as input to the `TransformerEncoder` class. In this class, we create a list of  $N$  layers by cloning the encoder layer with a deep copy and we register the parameters with a `ModuleList` as with the multihead attention in Sect. 15.3. In the original paper, Vaswani et al. (2017) proposed  $N = 6$ .

```
class TransformerEncoder(nn.Module):
    def __init__(self,
                 encoder_layer,
                 num_layers):
        super().__init__()
        self.encoder_stack = nn.ModuleList(
            [copy.deepcopy(encoder_layer)
             for _ in range(num_layers)])

    def forward(self, x):
        for encoder_layer in self.encoder_stack:
            x = encoder_layer(x)
        return x
```

**Fig. 15.8** Positional encoding and input. After Vaswani et al. (2017)



We create an encoder layer with:

```
>>> encoder_layer = TransformerEncoderLayer(d_model, nhead)
```

and then an encoder of  $N$  layers of `encoder_layer` objects with:

```
>>> num_layers = 6
>>> encoder = TransformerEncoder(encoder_layer, num_layers)
```

Finally, we call the encoder with:

```
>>> encoder(X)
```

to have the encoded embeddings.

## 15.8 Input Embeddings

Before we can apply the encoder, we must convert the input words to trainable embedding vectors. We first create index-to-word and word-to-index dictionaries from the vocabulary. We assign the unknown token to index 1 and the padding symbol to 0 as in Sect. 14.8:

```
idx2word = dict(enumerate(vocabulary, start=2))
word2idx = {word: idx for idx, word in idx2word.items()}
```

We then create a lookup table, `input_embeddings`, as in Sect. 14.8 that we fill with GloVe50 as initial values. We initialize the input embeddings tensor with random values:

```
input_embeddings = torch.rand(
    (len(word2idx) + 2, d_model))/10 - 0.05
```

and we replace the random row vectors with those of GloVe when they exist.

```
for word in embeddings_dict:
    input_embeddings[word2idx[word]] = embeddings_dict[word]
```

## 15.9 Positional Encodings

In Sect. 9.5.3, we presented bag-of-words techniques, where the word order did not matter in the vectorization of a document or a sentence. So far, our encoder has the same property and weakness. It is obvious that the order of the words in a sentence conveys a part of its meaning and hence that bags of words miss a part of this semantics.

Vaswani et al. (2017) proposed two techniques to add information on the word positions. Both consist of vectors of dimension  $d_{\text{model}}$  that are summed with the input embeddings:

1. The first one consists of trainable position embeddings, i.e. index  $i$  is associated with a vector of dimension  $d_{\text{model}}$  that is summed with the embedding of the input word at index  $i$ ;
2. The other consists of fixed vectors encoding the word positions. For a word at index  $i$ , the vector coordinates are defined by two functions:

$$PE(i, 2j) = \sin\left(\frac{i}{10000^{\frac{2j}{d_{\text{model}}}}}\right),$$

$$PE(i, 2j + 1) = \cos\left(\frac{i}{10000^{\frac{2j}{d_{\text{model}}}}}\right).$$

Let us program this second function for an input sequence of length `max_len`. The output data structure will be a matrix of `max_len` rows and  $d_{\text{model}}$  columns, where a row at index  $i$  contains the embedding vector at this position. We compute the angle of the sine and cosine by breaking it down into a dividend  $i$ , representing the word index, and a divisor,  $10000^{\frac{2j}{d_{\text{model}}}}$ , representing a coordinate in the embeddings. We then compute the sine and cosine of these angles alternatively using the fancy indexing properties of PyTorch:

```
def pos_encoding(max_len, d_model):
    dividend = torch.arange(max_len).unsqueeze(0).T
    divisor = torch.pow(10000.0,
                        torch.arange(0, d_model, 2)/d_model)
    angles = dividend / divisor
    pe = torch.zeros((max_len, d_model))
    pe[:, 0::2] = torch.sin(angles)
    pe[:, 1::2] = torch.cos(angles)
    return pe
```

We create the positional encoding vectors with a maximum sentence length with:

```
max_len = 100
pos_embeddings = pos_encoding(max_len, d_model)
```

## 15.10 Converting the Input Words

We can now create an `Embedding` class to convert the indices of the input words to embeddings and add the positional encoding. In the `__init__()` method, we create the embedding tables. We set the positional encoding as untrainable. In the `forward()` method, we just add the input and positional embeddings. Following Vaswani et al. (2017), we multiply the input embeddings by  $\sqrt{d_{\text{model}}}$  and we apply a dropout to the sum of embeddings.

```

class Embedding(nn.Module):
    def __init__(self,
                 vocab_size,
                 d_model,
                 dropout=0.1,
                 max_len=500):
        super().__init__()
        self.d_model = d_model
        self.input_embedding = nn.Embedding(vocab_size, d_model)
        pe = self.pos_encoding(max_len, d_model)
        self.pos_embedding = nn.Embedding.from_pretrained(
            pe, freeze=True)
        self.dropout = nn.Dropout(dropout)

    def forward(self, X):
        pos_mat = torch.arange(X.size(dim=-1))
        X = self.input_embedding(X) * math.sqrt(self.d_model)
        X += self.pos_embedding(pos_mat)
        return self.dropout(X)

    def pos_encoding(self, max_len, d_model):
        dividend = torch.arange(max_len).unsqueeze(0).T
        divisor = torch.pow(10000.0,
                            torch.arange(0, d_model, 2)/d_model)
        angles = dividend / divisor
        pe = torch.zeros((max_len, d_model))
        pe[:, 0::2] = torch.sin(angles)
        pe[:, 1::2] = torch.cos(angles)
        return pe

```

We create an embedding object with:

```

>>> vocab_size = len(word2idx) + 2
>>> embedding = Embedding(vocab_size, d_model)

```

The input embeddings of this object are randomly initialized. We assign them pretrained values with:

```
>>> embedding.input_embedding.weight = nn.Parameter(input_embeddings)
```

We then generate the input indices from the list of words from *Odyssey*'s quote:

```

>>> x = torch.LongTensor(
    list(map(lambda x: word2idx.get(x, 1), words_o)))

```

As the `Embedding` class contains a dropout, it will drop samples in the training mode. In the evaluation mode, it will keep all the samples. This is what we want here and we apply it to the indices:

```

>>> embedding.eval()
>>> X = embedding(x)

```

`X` is the input to the encoder:

```
>>> Y = encoder(X)
```

that creates the output vectors from the input embeddings.

## 15.11 Improving the Encoder

In this chapter so far, we have described all the elements we need to build the encoder part of a transformer. This enabled us to understand its structure more deeply. To simplify their implementation, we have created functions that apply to one sample, i.e. one sentence. In a real application, among the possible improvements to our code, we would need to process minibatches stored in 3rd order tensors (sometimes called 3D matrices), where the first axis corresponds to the samples, the second to the word indices, and the third one to the embeddings.

To see how we would proceed with a minibatch, let us integrate our two sentences from the *Odyssey* and Amazon.com in the  $X$  matrix. We first create a list of indices with:

```
sentences = [words_o] + [words_a]
sent_idx = []
for sent in sentences:
    sent_idx += [torch.LongTensor(
        list(map(lambda x: word2idx.get(x, 1),
                sent)))]
```

and we pad them with the statements:

```
>>> from torch.nn.utils.rnn import pad_sequence
>>> X_idx = pad_sequence(sent_idx, batch_first=True)
```

yielding the input tensor:

```
tensor([[ 43,  392,  244,  139,      6,   194, 1372,      7,
         6,   194, 1696],
       [ 55,  548,      7, 1372,  394,   462,      0,      0,
         0,      0,      0]])
```

We create the batched embeddings with

```
>>> X_batch = embedding(X_idx)
```

and we check that the embedding tensor consists of two samples of 11 aligned word indices, where each word is represented by a 50-dimensional embedding:

```
>>> X_batch.size()
torch.Size([2, 11, 50])
```

However, we cannot apply our encoder stack, `encoder()`, to this new  $X_{\text{batch}}$  tensor as we computed the self-attention with a matrix product. The operation:

```
X_batch @ X_batch.T
```

throws an error. Instead, we must use a batched matrix multiplication, `bmm(a, b)`, that computes the products of the pairs of matrices in `a` and `b`. We also transpose the index and embedding axes of the second term:

```
torch.bmm(X_batch,
          torch.transpose(X_batch, 1, 2))
```

The respective matrix products are stored in a batch that we can normalize with a softmax function and further multiply with  $X_{\text{batch}}$ . We would then pass on the results to the rest of the stack. Nonetheless, the complete adaptation of the encoder to batches is left as an exercise.

## 15.12 PyTorch Transformer Modules

Fortunately for us, PyTorch has a set of optimized modules to implement transformers that can handle minibatches. We describe them now. Overall, they use the same names as those in the previous sections and, for an elementary input, the same parameters. We just add `nn.` to refer to a PyTorch module. Of course, these modules have many more options.

In a minibatch, the index matrices are aligned to the longest sentence and thus include padding tokens for the others. We have to remove these tokens from the attention mechanism. We carry this out with a masking matrix, where we set the elements to true when we have a padding symbol. For our two sentences, we have:

```
>>> padding_mask = (X_idx == 0)
>>> padding_mask
tensor([[False, False, False, False, False, False, False,
        False, False, False, False],
       [False, False, False, False, False, False, True,
        True, True, True, True]])
```

### 15.12.1 The PyTorch Multihead Attention Class

The first module is a multihead attention class. We create a multihead attention object with this line:

```
multihead_attn = nn.MultiheadAttention(d_model,
                                       nhead,
                                       batch_first=True)
```

here with five heads and for the GloVe 50-dimensional embeddings.  $d_k$  has a value that follows the rule in Sect. 15.3:  $d_{\text{model}} // \text{nhead}$ . We use batches with the conventional order, where the rows correspond to the input words, and we need to set `batch_first` to true.

We apply the attention to the input vectors with:

```
attn_output, attn_weights = multihead_attn(Q, K, V)
```

and a `key_padding_mask` argument. In the case of self-attention, `Q`, `K`, and `V` are the same tensor as in Sect. 15.2.3, here  $X_{\text{batch}}$ . For our two sentences, this is then:

```
attn_output, attn_weights = multihead_attn(
    X_batch, X_batch, X_batch, key_padding_mask=padding_mask)
```

`attn_output` contains the output embeddings and `attn_weights` are averaged across the heads by default. The  $W^Q$ ,  $W^K$ ,  $W^V$ , and  $W^O$  matrices are randomly initialized. We can inspect them with the `state_dict()` method:

```
>>> multihead_attn.state_dict()
OrderedDict([('in_proj_weight',
              tensor([[ -0.0294, -0.1154, ...,
              ('in_proj_bias',
              tensor([0., 0., 0., ...,
              ('out_proj.weight',
              tensor([[ 0.1335,  0.1068, -0.0705, ....
              ('out_proj.bias',
              tensor([0., 0., 0., ...)
```

### 15.12.2 The PyTorch TransformerEncoder Class

PyTorch has an optimized class to create a fully functional encoder layer that incorporates the multihead attention from the previous section, `nn.TransformerEncoderLayer`:

```
encoder_layer = nn.TransformerEncoderLayer(d_model,
                                            nhead,
                                            batch_first=True)
```

and another one to create a stack of  $N$  identical layers, `nn.TransformerEncoder`:

```
encoder = nn.TransformerEncoder(encoder_layer, num_layers)
```

With these classes, creating the core of an encoder only takes two lines of Python and is thus extremely easy.

We obtain the output vectors for one layer with:

```
enc_layer_output = encoder_layer(
    X_batch, src_key_padding_mask=padding_mask)
```

and for the complete encoder stack with:

```
enc_output = encoder(X_batch,
                     src_key_padding_mask=padding_mask)
```

## 15.13 Application: Sequence Annotation

The input and output sequences of an encoder transformer have equal length. A first application is then to use it as an annotator for POS tagging or named entity recognition as in Chap. 14, *Part-of-Speech and Sequence Annotation*. Given an embedding layer and an encoder stack, the output consists of vectors of  $d_{\text{model}}$  dimensionality. We just need to add a linear layer to carry out a classification into tags.

We encapsulate `nn.TransformerEncoderLayer` and `nn.TransformerEncoder` in a class and we add a linear module.

```
class Model(nn.Module):
    def __init__(self,
                 embeddings,
                 d_model,
                 nhead,
                 num_layers,
                 nbr_classes,
                 dim_feedforward=2048,
                 dropout=0.1):
        super().__init__()
        self.embeddings = embeddings
        self.encoder_layer = nn.TransformerEncoderLayer(
            d_model,
            nhead,
            batch_first=True,
            dropout=dropout,
            dim_feedforward=dim_feedforward)
        self.encoder = nn.TransformerEncoder(
            self.encoder_layer, num_layers)
        self.fc = nn.Linear(d_model, nbr_classes)

    def forward(self, X):
        padding_mask = (X == 0)
        X = self.embeddings(X)
        X = self.encoder(
            X, src_key_padding_mask=padding_mask)
        return self.fc(X)
```

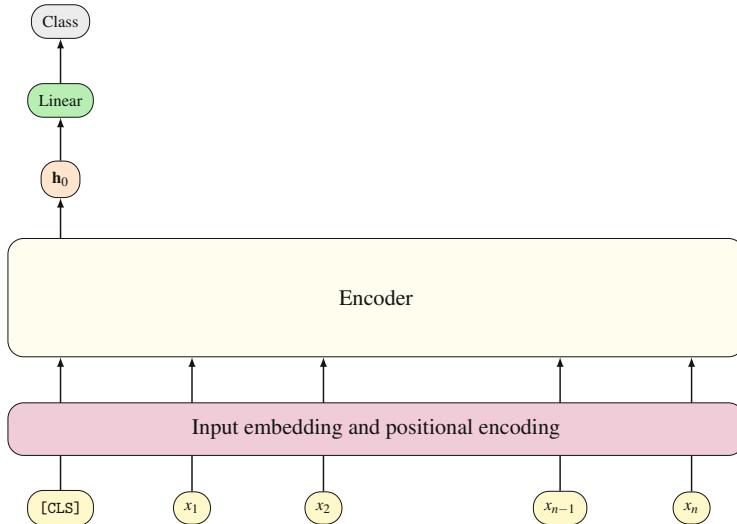
We create an input and positional embedding object as in Sect. 15.10 with the same class. We initialize the input embeddings with the GloVe values in `embedding_table`:

```
embedding = Embedding(vocab_size, d_model, dropout=dropout)
embedding.input_embedding.weight = nn.Parameter(embedding_table)
```

We pass this embedding object to the model when we create it:

```
model = Model(embedding,
              d_model,
              nhead,
              num_layers,
              NB_CLASSES + 1,
              dim_feedforward=dim_feedforward,
              dropout=dropout)
```

The rest of the program: The dataset preparation and training loop are identical to that in Sect. 14.8. Training the encoder and evaluating it on CoNLL 2003 data, we will not reach the scores we obtained with LSTMs however. We will see how we can improve them with pretraining in the next chapter.



**Fig. 15.9** Sentence classification with an encoder

## 15.14 Application: Classification

We can also apply our encoder to text classification. As we have here just one output, the class of the text, we will prefix the input sequence with a specific start token  $[CLS]$ , see Fig. 15.9. We will then use the encoded vector of it,  $h_0$  on the figure, as input to a linear layer. The multihead attention that collects influences from all the tokens will enable it to integrate the semantics of the sentence. Using a softmax function, we will then output the class. This technique is similar to that of the BERT system (Devlin et al. 2019) that we will describe in the next chapter.

To implement this idea, we just need to modify the last line of the previous class:

```
return self.fc(X)
```

and the  $X$  matrix to extract the first output vector in the sentences of the batch and pass them to the linear layer:

```
return self.fc(X[:, 0, :])
```

## 15.15 Further Reading

Self-attention and the transformer architecture immediately attracted a considerable interest when Vaswani et al. introduced them in 2017. The authors reported a significant improvement of translation performance with the complete encoder-decoder architecture and, soon after, Devlin et al. (2019) surpassed the state-of-the-art in

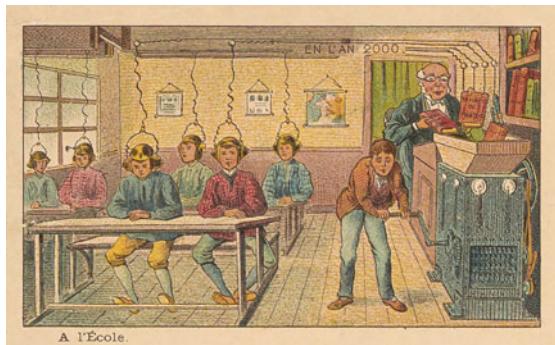
classification with the encoder alone. This architecture encountered an extraordinary success and spurred an uncountable number of variants. This makes it the dominant model in many areas of NLP today; see Lin et al. (2022) for a survey.

Scores of programmers reimplemented the transformer architecture. Among them, *The Annotated Transformer* by Rush (2018) is an interesting read as it includes the text of the original paper and a line-by-line implementation in PyTorch and a Python notebook.

Tensor libraries like PyTorch or Keras have excellent implementations of transformers. Their documentation and, if needed, their source code is a highly valuable source of information. Finally, Hugging Face (Wolf et al. 2020) is a repository of open-source implementations of transformers.

# Chapter 16

## Pretraining an Encoder: The BERT Language Model



*En l'an 2000, À l'École* by Jean-Marc Côté? (Source: [https://commons.wikimedia.org/wiki/File:France\\_in\\_XXI\\_Century.\\_School.jpg](https://commons.wikimedia.org/wiki/File:France_in_XXI_Century._School.jpg))

So far, outside of the embeddings, where we initially used GloVe, the encoder stack is filled with random parameters that we trained on CoNLL 2003, a relatively small annotated dataset. Devlin et al. (2019) proposed a method to pretrain models on very large raw corpora that found applications in the whole spectrum of language processing applications: The bidirectional encoder representations from transformers (BERT).

To fit the parameters, BERT proceeds in two steps:

1. The first step essentially boils down to training a language model on a large dataset. The language model predicts masked words inside a sentence and if two sentences follow each other. This step uses a raw corpus, needing no annotation. It enables the model to learn associations between the words. This step is called the pretraining;

2. The second step adapts the model to a specific application, for instance text classification or named entity recognition. It requires an annotated dataset, usually much smaller, where it will adjust further and more finely the parameters. This step is called the fine-tuning.

The two main pillars of the first step are the corpus gathering and preparation and the creation of a language model. We describe them now and, to exemplify the procedures, we will use the two first abridged sentences of the *Odyssey*:<sup>1</sup>

Tell me, O Muse, of that ingenious hero  
Many cities did he visit

## 16.1 Pretraining Tasks

In Chap. 10, *Word Sequences*, we designed a language model that, given a sequence of words, predicts the word to follow. The BERT language model is a bit different. It draws its inspiration from cloze tests (Taylor 1953) that we already saw in Sect. 11.9.

The pretraining step consists of two simultaneous classification tasks, where each input sample is a pair of sentences (in fact word sequences):

1. The masked language model (MLM), for which we replace some tokens of the input sample with a specific mask token [MASK]. We train the model to predict the value of the masked words as in this example restricted to one sentence:

```
Input: Tell me , O [MASK] , of that ingenious [MASK]
Predictions:           Muse                      hero
```

2. The next sentence prediction (NSP), where we create pairs of sentences that may or may not follow each other. Each pair corresponds to an input sample and we train the model to predict if the second sentence is the successor of the first one or not. Below are two examples with the two classes, where possibly masked words are written in clear:

```
Sentence pair:
(1) Tell me , O Muse , of that ingenious hero
(2) Many cities did he visit
Is next: True

Sentence pair:
(1) Tell me , O Muse , of that ingenious hero
(2) Exiled from home am I ;
Is next: False
```

---

<sup>1</sup> The complete quote is: *Tell me, O Muse, of that ingenious hero who travelled far and wide after he had sacked the famous town of Troy. Many cities did he visit, and many were the nations with whose manners and customs he was acquainted;*

## 16.2 Creating the Dataset

Devlin et al. (2019) created pairs of sentences with masked words to match precisely these two tasks. As pretraining corpus, BERT used the English Wikipedia, excluding the tables and figures, and BooksCorpus, totaling about 3.3 billion words. They formatted the corpus with one sentence per line and an empty line between the documents. They tokenized the text with WordPiece, see Chap. 13, *Subword Segmentation*, limiting the vocabulary to 30,000 tokens.

We will now write a small program to materialize these ideas. The complete implementation is available from Google Research GitHub repository.<sup>2</sup>

### 16.2.1 Input Sequence

Assuming we have read the corpus, split it into sentences, and formatted the dataset in a list of sentences as:

```
sentences = [
    'Tell me, O Muse, of that ingenious hero',
    'Many cities did he visit']
```

For each pair of sentences denoted  $A$  and  $B$ , we build two input sequences: the list of tokens and the list of segment identifiers.

1. We concatenate the list of tokens of the two segments  $A$  and  $B$ . We insert two special tokens to mark their boundaries: **[CLS]** at the start of the first segment and **[SEP]** at the end of both segments. Considering our two sentences and setting aside the subword tokenization, this yields the markup:

**[CLS]** Tell me , O Muse , of that ingenious hero **[SEP]** Many cities did he visit **[SEP]**

2. In the segment list, we replace each word in segment  $A$  with 0 and in segment  $B$  with 1.

Below is the code to implement this, where we store the pairs in a list of dictionaries. A dictionary contains three keys so far: **tokens**, the list of tokens, **segment\_ids** the list of segment markers, and **is\_next**, a Boolean telling if the two segments follow each other:

```
def create_sample(tokens_a, tokens_b, next=True):
    tokens = ['[CLS]'] + tokens_a + ['[SEP]']
    segment_ids = len(tokens) * [0]
    tokens.extend(tokens_b + ['[SEP]'])
    segment_ids += (len(tokens_b) + 1) * [1]
    sample = {'tokens': tokens,
```

---

<sup>2</sup> <https://github.com/google-research/bert/tree/master> in the [http://org.doi/create\\_pretraining\\_data.py](http://org.doi/create_pretraining_data.py) file.

```

        'segment_ids': segment_ids,
        'is_next': True}
    if not next:
        sample['is_next'] = False
    return sample

```

We tokenize and create the samples with:

```

import regex as re
pat = r'\p{P}|\[^\\s\\p{P}\]+'

tokenized_sents = []
for sent in sentences:
    tokenized_sents += [re.findall(pat, sent)]

dataset = []
for i in range(len(tokenized_sents) - 1):
    sample = create_sample(tokenized_sents[i],
                           tokenized_sents[i + 1])
    dataset += [sample]

```

For our tiny dataset, this yields:

```
[{'tokens': '[CLS]', 'Tell', 'me', ',', 'O', 'Muse', ',',
 'of', 'that', 'ingenious', 'hero', '[SEP]', 'Many',
 'cities', 'did', 'he', 'visit', '[SEP]'],
 'segment_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 1, 1, 1, 1, 1],
 'is_next': True}]
```

### 16.2.2 Next Sentence Prediction

BERT’s pretraining input consists of pairs of segments. In half of these pairs, the sentences follow each other, as in our example. In the other half, the second sentence is randomly sampled from another document. In BERT’s original implementation, the length of the input is of exactly 128 or 512 subword tokens. To create the pairs, the complete algorithm is:

1. Append consecutive sentences of the corpus until we reach this length or surpass it;
2. Split randomly the result into two segments,  $A$  and  $B$ , so that each segment consists of a nonempty sequence of sentences;
3. In half of the pairs, the second segment should not be the successor of the first one. For these samples, replace the second segment with sentences from another document;
4. Trim excess tokens at the front and end of the longest segment to match the length limit.

This algorithm is more complex than our examples in this chapter. We leave its implementation to the reader.

Once we have a dataset with consecutive and nonconsecutive segments, we can train a model. The next sentence prediction task is a binary classification with the `is_next` Boolean set to `True` or `False`. It is designed to learn relationships between sentences. For this, we add a linear layer on top of the encoder and we use the encoded value of the '`[CLS]`' token to predict `is_next` as in Sect. 15.14 and Fig. 15.9.

### 16.2.3 Masked Language Model

This MLM task is a cloze test, where Devlin et al. (2019) randomly selected 15% of the words for masking with at least one masked word per sample. It also applies to the pairs of segments and for our two sentences, this would correspond to 3 words as we have 15 tokens, for instance:

```
[CLS] Tell me , O [MASK] , of that ingenious [MASK] [SEP] Many [MASK] did he visit  
[SEP]
```

The program below tokenizes the sentences and replaces 15% of the tokens with '`[MASK]`'. We add two keys to our dictionaries: `masked_pos`, the positions of the masked words, and `masked_tokens`, their value.

```
def mask_tokens(sample,  
               mask_prob=0.15,  
               max_predictions=20):  
    cand_idx = [  
        i for i in range(len(sample['tokens']))  
        if sample['tokens'][i] not in ['[CLS]', '[SEP]'])  
    mask_cnt = max(1, int(round(len(sample['tokens'])  
                           * mask_prob)))  
    mask_cnt = min(mask_cnt, max_predictions)  
    random.shuffle(cand_idx)  
    sample['masked_pos'] = sorted(cand_idx[:mask_cnt])  
    sample['masked_tokens'] = [  
        token for i, token in  
        enumerate(sample['tokens'])  
        if i in sample['masked_pos']]  
    sample['tokens'] = [  
        '[MASK]' if i in sample['masked_pos'] else token  
        for i, token in enumerate(sample['tokens'])]  
    return sample
```

Applying this function to our dataset, we have:

```
>>> dataset  
[{'tokens': ['[CLS]', 'Tell', 'me', ',', 'O', '[MASK]', ',',  
          'of', 'that', 'ingenious', '[MASK]', '[SEP]', 'Many',  
          '[MASK]', 'did', 'he', 'visit', '[SEP]'],
```

```
'segment_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1],
'is_next': True,
'masked_pos': [5, 10, 13],
'masked_tokens': ['Muse', 'hero', 'cities']]}
```

With this procedure, the model could rely too heavily on the `[MASK]` token for predictions in other applications. To remove this dependency, in the complete implementation, out of the 15% selected tokens:

- 80% of the time, the program keeps the replacement as it is with `[MASK]`;
- 10% of the time, it replaces the mask with random words;
- 10% of the time, it keeps the original word.

## 16.3 The Input Embeddings

Once we have created the pairs of sentences, we map each token to the sum of three trainable embedding vectors: the token embedding, the segment embedding, and the positional embedding of the token in the sentence:

- Each token in the vocabulary, 30,000 WordPiece subwords in BERT base, is associated with a specific embedding vector;
- For the segments, we have two embedding vectors representing either the first or second sentence;
- Each position index in the sequence is associated with a learned embedding vector. The combined length of both sentences is less than 512, hence there are 512 positional embeddings. Note that this is a difference with the constant positional encoding of the encoder in Sect. 15.9.

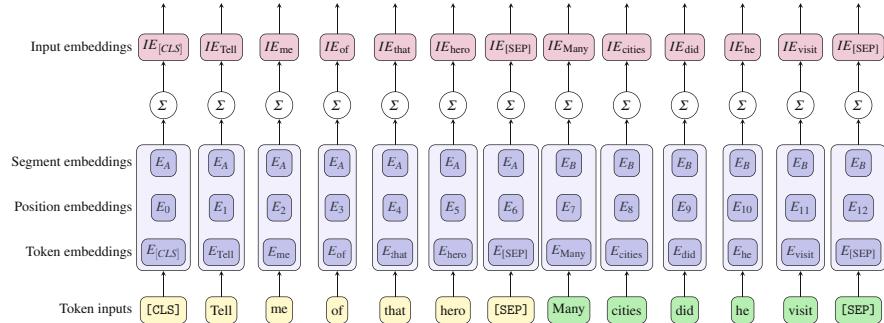
In the BERT base version, each embedding is a 768-dimensional vector.

Figure 16.1 shows the input with the two sentences simplified from the previous section:

Tell me of that hero. Many cities did he visit.

where we stripped the punctuation. For each token, BERT looks up the three corresponding vectors, token, segment, and position, and sums them to get the input representation. The two first vectors in Fig. 16.1 are:  $E_{CLS} + E_A + E_0$  and  $E_{Tell} + E_A + E_1$ . The final input representation for the 13 words is a matrix of size  $13 \times 768$ , one vector (row) per word.

To get the word embeddings, we use the PyTorch `Embedding` class. We need first to convert the tokens into indices as well as five special tokens: `['CLS']`, `['UNK']`, `['PAD']`, `['SEP']`, and `['MASK']`. We extract the vocabulary, `vocabulary`, the vocabulary size, `vocab_size`, the word-to-index and index-to-word dictionaries. This is something we have done multiple time in this book and we skip this piece of code. We set the padding index to 0. To store them, we add two last keys to our dictionaries: `token_ids` and `masked_ids`.



**Fig. 16.1** The BERT input consists of two segments bounded by the [CLS] and [SEP] symbols. The input embeddings are the sum of three trainable embeddings: Token, segment, and position. Each embedding is a  $d_{\text{model}}$ -dimensional vector, where  $d_{\text{model}} = 768$  with BERT

Computing these indices from Homer's work and applying the word-to-index conversion, we have for our small dataset:

```
>>> dataset
[{'tokens': '[CLS]', 'Tell', 'me', ',', 'o', '[MASK]', ',',
 'of', 'that', 'ingenious', '[MASK]', '[SEP]', 'Many',
 '[MASK]', 'did', 'he', 'visit', '[SEP']'],
 'token_ids': [1897, 1688, 6051, 7, 1209, 1898, 7, 6428,
 8726, 5463, 1898, 1899, 1064, 1898, 3680, 5075, 9308, 1899],
 ...}]
```

Let us now write a class to represent these embeddings. The forward method sums the vectors and, following Devlin et al. (2019), we add a dropout of 10% and we apply a layer normalization.

```
class BERTEmbedding(nn.Module):
    def __init__(self, vocab_size,
                 d_model=d_model,
                 maxlen=512,
                 n_segments=2,
                 dropout=0.1):
        super().__init__()
        self.d_model = d_model
        self.tok_embedding = nn.Embedding(vocab_size, d_model,
                                         padding_idx=0)
        self.pos_embedding = nn.Embedding(maxlen, d_model)
        self.seg_embedding = nn.Embedding(n_segments, d_model)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, token_ids, segment_ids):
        pos_ids = torch.arange(token_ids.size(dim=1),
                               dtype=torch.long).unsqueeze(0)
        embedding = self.tok_embedding(token_ids) + \
                   self.pos_embedding(pos_ids) + \
```

```
        self.seg_embedding(segment_ids)
    return self.layer_norm(self.dropout(embedding))
```

We create an embedding object and we apply it to our sequence with:

```
embeddings = BERTEmbedding(vocab_size)
embeddings(dataset[0]['token_ids'].unsqueeze(0),
           dataset[0]['segment_ids'].unsqueeze(0))
```

This results in a tensor of size `torch.Size([1, 18, 768])` as we have one sample, 18 tokens, and  $d_{\text{model}} = 768$ .

## 16.4 Creating a Batch

The training input consists of batches of sequences. We create a function to extract all the input parameters from the dataset. For the trainable embeddings, we need the token and segment indices. We generate the position indices in `BERTEmbedding` class. For the predictions, we need to know if the second segment is a successor or not of the first one as well as the position of the masked words and their values: `masked_pos` and `masked_ids`.

Differently to BERT, in our examples, sequences have different length. We pad the token and sequence identifiers with a padding symbol to create input tensors. Their default order in the PyTorch padding function and encoder transformers is (`sequence, batch, d_model`). We prefer the common convention of having the batch index first and we change the order with the argument: `batch_first=True`.

```
def make_batch(dataset, start, BATCH_SIZE=BATCH_SIZE):
    tok_ids = []
    seg_ids = []
    masked_pos = []
    masked_ids = []

    y_nsp = []
    if len(dataset) < start + BATCH_SIZE:
        end = len(dataset)
    else:
        end = start + BATCH_SIZE
    for i in range(start, end):
        tok_ids += [dataset[i]['token_ids']]
        masked_pos += [dataset[i]['masked_pos']]
        masked_ids += [dataset[i]['masked_ids']]
        seg_ids += [dataset[i]['segment_ids']]
        if dataset[i]['is_next']:
            y_nsp += [1]
        else:
            y_nsp += [0]

    y_nsp = torch.LongTensor(y_nsp)
    tok_ids = torch.nn.utils.rnn.pad_sequence(tok_ids,
```

```

                batch_first=True)
seg_ids = torch.nn.utils.rnn.pad_sequence(seg_ids,
                                         batch_first=True)
return tok_ids, seg_ids, y_nsp, masked_pos, masked_ids

```

## 16.5 BERT Architecture

In its base version, BERT’s architecture consists of 12 layers, each layer with 8 self-attention heads. The embedding dimension is 768 and must be divisible by the number of heads. Using PyTorch classes from Sect. 15.12, we can easily implement it in the `__init__()` method. The `forward()` method has two arguments: the sequence of tokens and the sequence of segment identifiers that are needed to compute the embeddings.

As the sequences are of unequal lengths, we have aligned them with padding symbols in the batch. We create a mask of Boolean values, `input_mask`, to indicate where the padding is. We tell the encoder to ignore it with the `src_key_padding_mask` argument. This is necessary to remove the corresponding symbols from the attention mechanism.

```

class BERT(nn.Module):
    def __init__(self, vocab_size,
                 d_model=768,
                 maxlen=128,
                 n_segments=2,
                 nhead=8,
                 num_layers=12):
        super().__init__()
        self.embeddings = BERTEmbedding(vocab_size,
                                         d_model,
                                         maxlen,
                                         n_segments)
        self.encoder_layer = nn.TransformerEncoderLayer(
            d_model,
            batch_first=True,
            nhead=nhead,
            dim_feedforward=4 * d_model)
        self.encoder = nn.TransformerEncoder(
            self.encoder_layer,
            num_layers=num_layers)

    def forward(self, token_ids, segment_ids):
        padding_mask = (token_ids == 0)
        embeddings = self.embeddings(token_ids, segment_ids)
        y = self.encoder(embeddings,
                         src_key_padding_mask=padding_mask)
        return y

```

We create a BERT object with the default parameters with

```
bert = BERT(vocab_size)
```

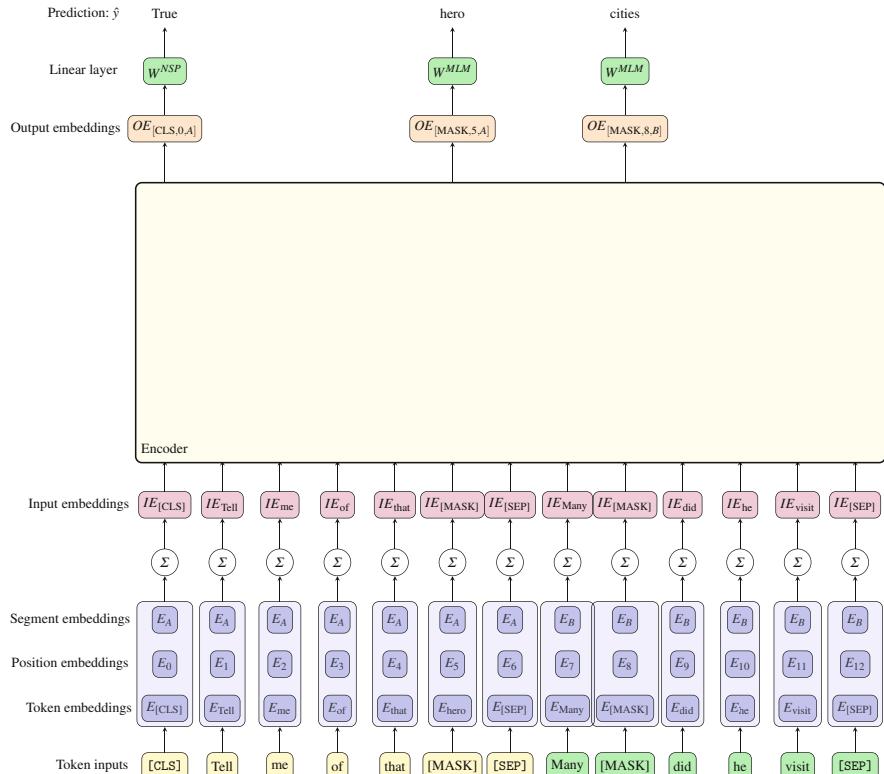
We then apply it to a batch with:

```
tok_ids, seg_ids, _, _, _ = make_batch(dataset, 0)
bert(tok_ids, seg_ids)
```

## 16.6 BERT Pretraining

Once we have built the BERT structure, we can train it with the masked language model and the next sentence prediction. We add two linear modules on top of the encoder, see Fig. 16.2:

1. One that will use the [CLS] output and predict if the second sentence is a successor of the first one or not;
2. A second that will read the output of the masked tokens and predict the corresponding word. In the original implementation, the masked language model



**Fig. 16.2** BERT pretraining: The [CLS] position is used to predict if the second sentence follows the first one and the [MASK] input to predict the actual word, here with a perfect results

uses the weights of the token embedding matrix as linear layer and adds trainable biases.<sup>3</sup>

The encoder outputs vectors for all the tokens in the sequence. We only consider the masked words only to compute the loss. We extract them from the encoder output with the `extract_rows()` method before we predict their values.

```
class BERTLM(nn.Module):
    def __init__(self, vocab_size,
                 d_model=768,
                 maxlen=128,
                 n_segments=2,
                 nhead=8,
                 num_layers=12):
        super().__init__()
        self.bert = BERT(vocab_size, d_model=d_model,
                         maxlen=maxlen,
                         n_segments=n_segments,
                         nhead=nhead,
                         num_layers=num_layers)
        self.masked_lm = nn.Linear(d_model, vocab_size)
        self.masked_lm.weight = self.bert.embeddings.
            tok_embedding.weight
        self.next_sentence = nn.Linear(d_model, 2)

    def forward(self, token_ids,
                segment_ids,
                masked_pos):
        y = self.bert(token_ids, segment_ids)
        y_lm = self.extract_rows(y, masked_pos)
        y_lm = self.masked_lm(y_lm)
        y_nsp = y[:, 0, :] # we extract CLS
        y_nsp = self.next_sentence(y_nsp)
        return y_lm, y_nsp

    def extract_rows(self, matrix, row_ids):
        y_lm_pred = []
        for x in range(matrix.size(dim=0)):
            y_lm_pred += [matrix[x, row_ids[x]]]
        return torch.cat(y_lm_pred)
```

We create a language model object with:

```
bert_lm = BERTLM(vocab_size)
```

and we apply it to a batch with:

```
tok_ids, seg_ids, y_nsp, masked_pos, masked_ids = \
    make_batch(dataset, 0)
Y_lm, Y_nsp = bert_lm(tok_ids, seg_ids, masked_pos)
```

---

<sup>3</sup> See: [https://github.com/google-research/bert/blob/master/run\\_pretraining.py](https://github.com/google-research/bert/blob/master/run_pretraining.py), line 240.

## 16.7 The Training Loop

For both predictions, the loss is the cross entropy and, for a batch of input sequences, we sum the mean of the next sentence prediction loss and the mean of the masked language model loss. We compute this cross entropy loss with `nn.CrossEntropyLoss()`.

We create the loss and the optimizer:

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(bert_lm.parameters(), lr=0.01)
```

And we write a training loop:

```
for epoch in range(60):
    random.shuffle(dataset)
    loss_sum_lm = 0
    loss_sum_nsp = 0
    bert_lm.train()
    for i in tqdm(range(n_batches)):
        t_ids, s_ids, y_nsp, masked_pos, masked_ids = make_batch(
            dataset, BATCH_SIZE * i)
        y_lm_pred, y_nsp_pred = bert_lm(
            t_ids, s_ids, masked_pos)
        masked_ids = torch.LongTensor(
            [id for ids in masked_ids for id in ids])
        loss_nsp = loss_fn(y_nsp_pred, y_nsp)
        loss_lm = loss_fn(y_lm_pred, masked_ids)
        loss = loss_nsp + loss_lm
        loss_sum_lm += loss_lm.item()
        loss_sum_nsp += loss_nsp.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

## 16.8 BERT Applications

Devlin et al. (2019) applied the pretrained BERT encoder to a set of applications: text classification, classification of pairs of sentences, sequence tagging, and question answering. We first describe the classification of sentences and pairs of sentences.

### 16.8.1 Classification

As datasets for classification, Devlin et al. (2019) used the General Language Understanding Evaluation (GLUE) benchmark (Wang et al. 2018). GLUE has two sentence classification tasks and six regarding pairs of sentences:

- The first task is a binary classification. It uses two corpora, SST-2 (Socher et al. 2013) and CoLA (Warstadt et al. 2018), where the sentences in SST-2 are annotated with a positive or negative sentiment and those in CoLA, whether they are linguistically acceptable or not;
- The second task uses six corpora with pairs of sentences as input: MNLI (Williams et al. 2018), QQP (Iyer et al. 2017), QNLI (Wang et al. 2018), STS-B (Cer et al. 2017), MRPC (Dolan & Brockett 2005), and RTE (Bentivogli et al. 2009). MNLI labels each pair with either entailment, contradiction, or neutral; QQP and MRPC whether two questions or sentences are equivalent; STS-B whether two sentences are semantically equivalent with a score ranging from 1 to 5; and RTE whether the second sentence is an entailment of the first one. Finally, in the QNLI corpus, each pair consists of a question and a sentence. The pair is positive if the sentence contains the answer to the question and negative otherwise.

In the two first tasks, we classify a sentence,  $(x_1, x_2, \dots, x_n)$ , and in the six others, a pair of sentences,  $(x_1, x_2, \dots, x_n)$  and  $(x'_1, x'_2, \dots, x'_p)$ . Devlin et al. (2019) represented this input similarly to that in Sect. 16.2. They added a [CLS] token at the beginning of the sequence:

$$([\text{CLS}], x_1, x_2, \dots, x_n)$$

and, for the pairs, they also inserted a [SEP] token in between:

$$([\text{CLS}], x_1, x_2, \dots, x_n, [\text{SEP}], x'_1, x'_2, \dots, x'_p).$$

The classifier consists of the pretrained encoder model that they supplemented with an additional linear layer,  $W$ , on top of the encoder output of the [CLS] token,  $\mathbf{h}_0$  on Fig. 16.3. The vector of predicted probabilities is then given by:

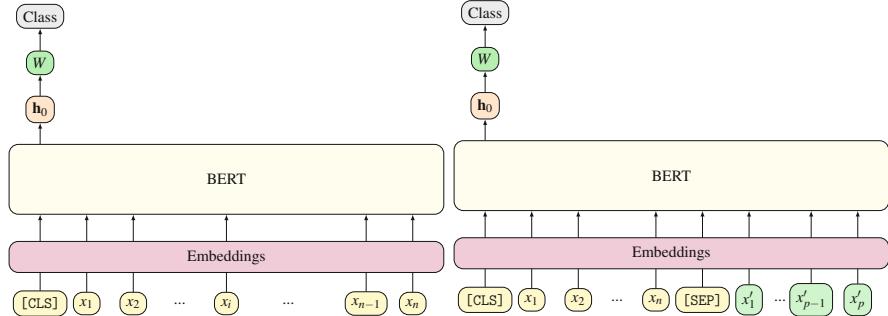
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{h}_0 W^\top).$$

For each corpus, they finetuned the classifier either by fitting the last layer weights,  $W$ , or the parameters of the complete encoder, including  $W$ . This fine-tuning task requires a much smaller corpus than the pretraining task and is much faster.

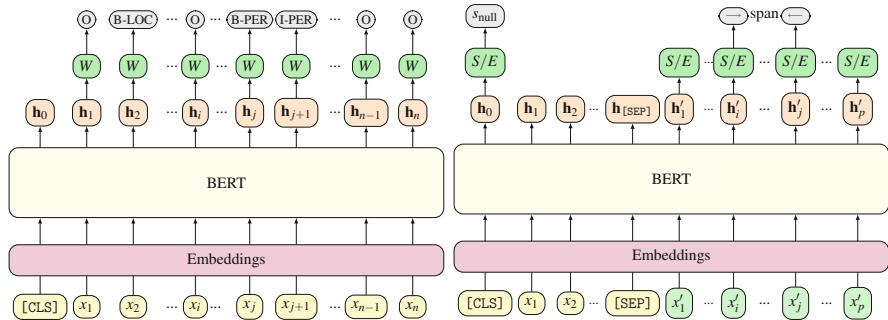
With their pretraining and fine-tuning procedure, Devlin et al. (2019) could outperformed other previous architectures and methods on the GLUE tasks.

### 16.8.2 Sequence Annotation

In Sect. 15.13, we used an encoder to annotate sequences. We can replace it with BERT as its structure is nearly identical. The main difference is that we use weights pretrained with the masked language model and next sentence prediction instead of



**Fig. 16.3** BERT applications: sentence classification, left part, and pair classification, right part



**Fig. 16.4** BERT applications: Sequence annotation, left part, and question answering, right part, where  $(x_1, x_2, \dots, x_n)$  corresponds to the question tokens and  $(x'_1, x'_2, \dots, x'_p)$  to the passage ones. The span  $(\mathbf{h}'_i, \dots, \mathbf{h}'_j)$  maximizes the  $S \cdot \mathbf{h}'_i + E \cdot \mathbf{h}'_j$  score and corresponds to the answer prediction  $(x'_i, \dots, x'_j)$

a random initialization. As input, we have a sequence of words:

$$([CLS], x_1, x_2, \dots, x_n)$$

that BERT encodes as a sequence of vectors of  $d_{\text{model}}$ -dimensionality:

$$(\mathbf{h}_0, \mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_n).$$

We apply a trainable linear layer  $W$  to classify it in a sequence of tags, for instance parts of speech or BIO named entity tags described in Sect. 14.10, such as B-PER, I-PER, B-LOC, I-LOC, etc. See Fig. 16.4, left part:

$$\hat{y}_i = \text{softmax}(\mathbf{h}_i W^\top).$$

We fine-tune the system as with classification in the previous section.

### 16.8.3 Question Answering

Finally, Devlin et al. (2019) applied BERT to a question answering task, more precisely to the Stanford question answering datasets, SQuAD 1.1 and 2.0 (Rajpurkar et al. 2016, 2018). SQuAD consists of pairs of questions and passages from Wikipedia as input and Rajpurkar et al. (2016) formulate question answering as the retrieval of an answer in the form of a text span in the input passage.

As an example, starting from the introductory paragraph of the article on *precipitation* in Wikipedia:

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals within a cloud. Short, intense periods of rain in scattered locations are called “showers.”

Rajpurkar et al. (2016) posed three questions for which the answers are underlined in the text:

1. *What causes precipitation to fall?* **gravity**;
2. *What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?* **graupel**;
3. *Where do water droplets collide with ice crystals to form precipitation?* **within a cloud**.

In SQuAD 1.1, the 100,000 questions always have an answer in their associated passage. This is not the case in SQuAD 2.0, where the authors added 50,000 unanswerable questions to it.

To fine-tune BERT, Devlin et al. (2019) formatted the question-passage pairs as in the next sequence prediction:

$$([\text{CLS}], x_1, x_2, \dots, x_n, [\text{SEP}], x'_1, x'_2, \dots, x'_p),$$

where  $x_1, x_2, \dots, x_n$  are the question tokens,  $x'_1, x'_2, \dots, x'_p$ , the passage tokens, and  $[\text{SEP}]$ , the separation symbol.

The SQuAD question answering resembles BIO sequence annotation in Sect. 14.10 as, given a question, we can mark the answer in the passage with begin and inside tags and the rest with outside tags. Devlin et al. (2019) chose to estimate two tags only, start and end, with two trainable weight matrices denoted  $S$  and  $E$  (Fig. 16.4, right part). In fact, as we have a binary classification,  $S$  and  $E$  are simply two vectors. The classification is then comparable to logistic regression with the dot products  $S \cdot \mathbf{h}'_i$  and  $E \cdot \mathbf{h}'_i$  to determine if index  $i$  is the start or end of the answer.

Logistic regression could yield multiple positive start and end tags in a passage. Instead Devlin et al. (2019) computed the dot products of the start position over the output sequence. They normalized them with the softmax function yielding a

probability distribution:

$$\hat{y}'_{\text{start}_i} = \frac{S \cdot \mathbf{h}'_i}{\sum_j S \cdot \mathbf{h}'_j}.$$

They did the same with the end:

$$\hat{y}'_{\text{end}_i} = \frac{E \cdot \mathbf{h}'_i}{\sum_j E \cdot \mathbf{h}'_j}.$$

They finally defined the score of a candidate span as  $s(i, j) = S \cdot \mathbf{h}'_i + E \cdot \mathbf{h}'_j$  with  $i \leq j$  and the answer prediction as the index pair maximizing:

$$(i_{\text{answer}}, j_{\text{answer}}) = \arg \max_{i \leq j} S \cdot \mathbf{h}'_i + E \cdot \mathbf{h}'_j.$$

The predicted answer is then the  $(x'_{i_{\text{answer}}}, \dots, x'_{j_{\text{answer}}})$  passage span.

Devlin et al. (2019) fine-tuned the model in Fig. 16.4, right part, with a loss set to the sum of the cross entropies of the correct  $S$  and  $E$  positions.

In SQuAD 2.0, some questions are unanswerable. Devlin et al. (2019) extended their model with a prediction of the encoded output of the [CLS] token,  $\mathbf{h}_0$ :

$$s_{\text{null}} = S \cdot \mathbf{h}_0 + E \cdot \mathbf{h}_0.$$

They predict an answerable question when

$$\max_{i \leq j} S \cdot \mathbf{h}'_i + E \cdot \mathbf{h}'_j > s_{\text{null}} + \tau,$$

where  $\tau$  is adjusted on the validation set so that it maximizes the final score.

Devlin et al. (2019) reported SQuAD F1 scores that outperformed all the other systems at the time they published their paper.

### 16.8.4 Question Answering Systems

The SQuAD benchmark alone does not reflect the entire complexity of a real question answering system. In this section, we outline a few practical points.

Before a system can answer any question, we must first collect a large set of documents that will form its knowledge source and store it in a repository. As with SQuAD, wikipedia is often a starting point for such a collection. Practically, we collect wikipedia articles with the scraping techniques described in Sect. 4.5. The size of the collection can range from a subset of articles to all of a wikipedia

language version, the whole wikipedia in all the languages, or include even more resources: either available from the web or nonpublic documents.

The wikipedia articles are sometimes quite long and thus impossible to process for a transformer. We usually split them into shorter paragraphs or passages. We then index them with the techniques in Sect. 9.5.1. Many question answering system use the Lucene<sup>4</sup> open-source tool for this. It is an excellent indexer that can process large quantities of text.

Transformer are too slow to examine the millions of documents sometimes needed to answer a question. That is why we must divide the answering process in two steps: The retrieval of a short list of passage candidates with a fast algorithm and then the identification of the answer in these passages. We call this combination the retriever-reader model:

1. Given a question, the passage retriever selects the paragraphs of the collection that may contain the answer and ranks them. The ranking algorithm represents the passages with vector space techniques, as in Sect. 9.5.3, or dense vectors, as in Chap. 11, *Dense Vector Representations*; we can use Sentence-BERT (SBERT) (Reimers and Gurevych 2019), for instance, to create sentence or paragraph embeddings; the passage retriever then computes the similarity between the question and a passage with cosines. Again, we can use Lucene for this or a vector database;
2. Using the short list of candidates from the previous step, the reader applies a classifier to decide if the answer is in the passage or not, and if yes, identifies it with the transformer.

## 16.9 Using a Pretrained Model

Pretraining a BERT model with the methods in Sects. 16.1 and 16.6 is beyond the computing capacity of many programmers. Fortunately, the BERT team made their pretrained models available and they were soon followed by many others. In this section, we will describe how to use a pretrained BERT model for a classification task. We will rely on the Hugging Face library (Wolf et al. 2020) and we will use the IMDB dataset of movie reviews (Maas et al. 2011).

To fine-tune a model, we need three components:

1. The model, for instance BERT, pretrained on a large unannotated corpus. The fine-tuning step will either reuse it as is, the parameters are said to be frozen, or refine its existing parameters;
2. The subword tokenizer that matches the tokens used to pretrain the model;
3. The classification head that projects the [CLS] output vector to the number of classes. This classification head will be trained from scratch.

---

<sup>4</sup> <https://lucene.apache.org/>.

Transformer models are very large. In this example, we will use a leaner version of BERT called DistilBERT (Sanh et al. 2019). DistilBERT has 40% less parameters than BERT based uncased and still reaches 95% of BERT’s performances on the GLUE benchmarks. This makes the experiment possible on a personal computer with no graphics processing unit (GPU).

## 16.10 The IMDB Dataset

Hugging Face provides a large repository of datasets that are compatible with their subsequent processing pipelines. The IMDB dataset consists of movie reviews annotated as positive or negative. We import it with these statements:

```
from datasets import load_dataset
imdb = load_dataset('imdb')
```

This downloads the dataset from a Hugging Face server and stores it in a cache folder such as: `~/.cache/huggingface/datasets`. Subsequent calls will load it from the cache.

Once dowloaded, `imdb` returns:

```
>>> imdb
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 25000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 25000
    })
    unsupervised: Dataset({
        features: ['text', 'label'],
        num_rows: 50000
    })
})
```

telling that the dataset consists of two annotated sets split into training and test and an unannotated set called unsupervised.

We assign the training and test sets with these statements

```
train_set = imdb['train']
test_set = imdb['test']
```

The negative and positive reviews are arranged as two blocks of equal lengths. We examine excerpts of the first and 12,500th reviews with:

```
>>> train_set[0]
{'text': 'I rented I AM CURIOUS-YELLOW from my video store...'
```

```

    But really, this film doesn't have much of a plot.',
    'label': 0}
>>> train_set[12500]
{'text': 'Zentropa has much in common with The Third Man, ...
    Like TTM, there is much inventive camera work...', 
    'label': 1}

```

where the first review is categorized as negative and the second as positive.

## 16.11 Tokenization

Each Hugging Face model has its tokenizer, BPE in the case of DistilBERT, trained on a large corpus and ready to use. We create it from the model name:

```

from transformers import AutoTokenizer

model_name = 'distilbert-base-uncased'
tokenizer = AutoTokenizer.from_pretrained(model_name)

```

These statements will download the tokenizer model the first time we execute them. Again, it will be stored in the cache. Alternatively, we can download the whole DistilBERT model explicitly, including the tokenizer, with the statement:

```
git clone https://huggingface.co/distilbert-base-uncased
```

We tokenize the first text from the dataset with:

```
tokens = tokenizer(train_set['text'][0])
```

where `tokens` has two keys:

```

>>> tokens.keys()
dict_keys(['input_ids', 'attention_mask'])

```

The `input_ids` are the token indices and the `attention_mask` are the valid tokens. Remember that in a batch, we have to pad the sequences to a same length to create a tensor. We mark the padded tokens with a 0 in `attention_mask`. Here there is only one sentence and all the tokens are valid.

```
{'input_ids': [101, 1045, 12524, 1045, 2572, ..., 102],
 'attention_mask': [1, 1, 1, 1, 1, ..., 1]}
```

We translate the indices to tokens with this statement:

```
>>> tokenizer.convert_ids_to_tokens(tokens['input_ids'])
['[CLS]', 'i', 'rented', 'i', ..., 'a', 'plot', '.', '[SEP]']
```

Let us now see the effects of the attention mask on this small corpus with sentences of different lengths:

```
classics = [
    'Tell me, O Muse, of that hero',
    'Many cities did he visit',
    'Exiled from home am I ;']
```

We specify the padding option to the tokenizer. As DistilBERT has a maximal sequence length of 512, and we ask it to truncate the sequences:

```
>>> tokenizer(classics, padding=True, truncation=True,
               max_length=512)

{'input_ids': [
[101, 2425, 2033, 1010, 1051, 18437, 1010, 1997, 2008, 5394, 102],
[101, 2116, 3655, 2106, 2002, 3942, 102, 0, 0, 0, 0],
[101, 14146, 2013, 2188, 2572, 1045, 1025, 102, 0, 0, 0]],
'attention_mask': [
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]]}
```

We tokenize the dataset and to speed up the process, we use the `Dataset.map` function. We set the maximal length to 128 to save memory:

```
imdb_tokenized = imdb.map(lambda x: tokenizer(x['text'],
                                              truncation=True,
                                              padding=True,
                                              max_length=128),
                           batched=True)
```

## 16.12 Fine-Tuning

### 16.12.1 Architecture

Now we can proceed to the fine-tuning task. The architecture consists of a pretrained model and a classification head. There are predesigned models from Hugging Face that just do that and that we will reuse. We just need to specify the number of classes, here two, to get the right output:

```
from transformers import AutoModelForSequenceClassification

num_labels = 2
model = (AutoModelForSequenceClassification
         .from_pretrained(model_name, num_labels=num_labels)
         .to('cpu'))
```

When printing the model, we see its structure starting with the embeddings, the transformer with six blocks, removed from the text here, and finally the classification head with two linear layers, `pre_classifier` and `classifier`:

```
DistilBertForSequenceClassification(
    (distilbert): DistilBertModel(
        (embeddings): Embeddings(
            (word_embeddings): Embedding(30522, 768, padding_idx=0)
            (position_embeddings): Embedding(512, 768)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
        )
        (transformer): Transformer(
            (layer): ModuleList(
                (0-5): 6 x TransformerBlock(
                    ...
                )
            )
            (pre_classifier): Linear(in_features=768, out_features=768, bias=True)
            (classifier): Linear(in_features=768, out_features=2, bias=True)
            (dropout): Dropout(p=0.2, inplace=False))
    )
)
```

### 16.12.2 Training Parameters

Before we start training, we need to specify the training arguments. The `TrainingArguments` class enables us to specify tons of them. Here we will mostly rely on those from Devlin et al. (2019)'s paper and the class defaults:

```
from transformers import TrainingArguments

model_output_name = 'finetuned_imdb'
training_args = TrainingArguments(output_dir=model_output_name,
                                  num_train_epochs=2,
                                  learning_rate=2e-5,
                                  per_device_train_batch_size=32,
                                  per_device_eval_batch_size=32,
                                  weight_decay=0.01,
                                  evaluation_strategy='epoch',
                                  disable_tqdm=False)
```

### 16.12.3 Training

We can now start the fine-tuning of the parameters. We create a **Trainer** object and we start it. Once trained, we save the model. With this statement, we also store the tokenizer in the model to be able to reuse it.

```
from transformers import Trainer

trainer = Trainer(model=model, args=training_args,
                  compute_metrics=compute_accuracy,
                  train_dataset=imdb_tokenized['train'],
                  eval_dataset=imdb_tokenized['test'],
```

```

        tokenizer=tokenizer)

trainer.train()
trainer.save_model()

```

We added a function to compute the accuracy of the model on the evaluation set, here the test set:

```

def compute_accuracy(eval_output):
    labels_ids = eval_output.label_ids
    predictions = eval_output.predictions.argmax(axis=-1)
    return {'accuracy': np.mean(predictions == labels_ids)}

```

This function has to be compatible with Hugging Face's `transformers.EvalPrediction` formats and returns a dictionary of metric values.

### 16.12.4 Prediction

We can run predictions with the model we have just trained. We will then need to apply the tokenizer before. The model input will consist of the input ids, the attention mask, and possibly the labels if we want to compute the loss:

```

finetuned_model = AutoModelForSequenceClassification.from_pretrained(
    'finetuned_imdb')

finetuned_model(input_ids, attention_mask=attention_mask, labels=labels)

```

We can also create a prediction pipeline that will include the tokenizer and the model:

```

from transformers import pipeline

ftuned_model_name = 'finetuned_imdb'
classifier = pipeline('text-classification',
                      model=ftuned_model_name)

>>> classifier(['I loved it!!!', 'I hate this movie!'])

[{'label': 'LABEL_1', 'score': 0.982995331287384},
 {'label': 'LABEL_0', 'score': 0.9783737659454346}]

```

### 16.12.5 Freezing Layers

When training a model, by default we fit all the parameters. In fact the different layers of a transformer learn different kinds of patterns, from very generic semantic relations to application-ready properties, where the lower layers are more abstract

and higher layers more applied. As a consequence, lower-level layers, if trained on a sufficiently large pre-training corpus, can be reused as is.

Even if DistilBERT is relatively small by today's standards, it has already a considerable number of parameters. We count them with this function:

```
def count_parameters(model):
    total = 0
    trainable = 0
    for params in model.parameters():
        total += params.numel()
        if params.requires_grad:
            trainable += params.numel()
    return {'total': total, 'trainable': trainable}

>>> count_parameters(model)
{'total': 66955010, 'trainable': 66955010}
```

We can posit that the accuracy of the IMDB classification will depend mostly on the last classification layers which have a much small number of parameters:

```
>>> count_parameters(model.pre_classifier)
{'total': 590592, 'trainable': 590592}

>>> count_parameters(model.classifier)
{'total': 1538, 'trainable': 1538}
```

By default, all the parameters are trainable. We “freeze” them with this loop that disables the gradients:

```
for param in model.parameters():
    param.requires_grad = False
```

We then set the two last classification layers as trainable with these loops:

```
for param in model.pre_classifier.parameters():
    param.requires_grad = True

for param in model.classifier.parameters():
    param.requires_grad = True

>>> count_parameters(model)
{'total': 66955010, 'trainable': 592130}
```

Loading the original pretrained model again and freezing the parameters as described, we create a new `Trainer` and fine-tune the classifier layers. After two epochs, this simpler setup yields accuracies that are well below that of the original one though: About 77.87% on the test set compared to 87.83% when training all the parameters.

## 16.13 Further Reading

BERT has been one of the first successful outcomes of transformers. It showed the capacity of large language models to encapsulate a massive amount of textual knowledge in a pipeline of matrices. Its excellent scores on multiple benchmarks proved its versatility. Instead of keeping their code secret, BERT’s authors made the implementation available on GitHub.<sup>5</sup> This ensured it an immense success soon followed by a multitude of replicas, extensions, or modifications: RoBERTa (Liu et al. 2019), Multilingual BERT, DistilBERT, etc. to name a few, either mono or multilingual.

Devlin et al. (2019, Sect. 5.2) showed that the encoder performance was linked to the size of the model: the larger, the better. This caused a race to gigantism with transformer models, all architectures included, now reaching a trillion parameters (Ren et al. 2023). Fitting such an astronomic number of parameters is not free however. Strubell et al. (2019) highlighted how much NLP models rely on the intensive use of electricity-hungry computing platforms and that they come at a considerable energy cost.

In this chapter, we fine-tuned and applied a text classification model with the Hugging Face application programming interface. Tunstall et al. (2022) describe in more detail this interface with application examples. In addition to IMDB and DistilBERT, the Hugging Face repository hosts numerous datasets, pretrained models, and ready-to-use classes to implement tasks such as the ones we described here, including sequence annotation and question answering. Due to its large number of open-source tools, Hugging Face has become a popular resource in the field of large language models.

---

<sup>5</sup> <https://github.com/google-research/bert/>.

# Chapter 17

## Sequence-to-Sequence Architectures: Encoder-Decoders and Decoders



Vaswani et al. (2017) designed the transformer architecture to transduce input sequences into output sequences, and more concretely, for machine translation. In fact, natural language processing was born with machine translation. Facing competition from Russia after the Second World War, the government of the United States decided to fund large-scale translation programs to have quick access to documents written in Russian. Machine translation has developed well beyond this objective and is now one of the most popular tool of the web.

In this chapter, machine translation will be our common thread. We will first describe parallel corpora, the raw material of machine translation, then a few techniques to align the sentences they contain. We will proceed with the description of the encoder-decoder architecture that complements the encoders we have seen in Chaps. 15, *Self-Attention and Transformers*, and 16, *Pretraining an Encoder: The BERT Language Model*, how we can build a concrete encoder-decoder with transformers, and how to program and train it with PyTorch to carry out translation. We will finally see how we can train and build applications with a standalone decoder architecture.

### 17.1 Parallel Corpora

Given the relatively long history of machine translation, a variety of methods have been experimented on and applied. Starting from the pioneering work of Brown et al. (1993), machine translation used statistical models and parallel corpora. We introduce them now.

Parallel corpora are the main data source of machine translation. Administrative or parliamentary texts of multilingual countries or organizations are widely used because they are easy to obtain and are often free. The Canadian Hansard or the European Parliament proceedings are examples of them. Table 17.1 shows an

**Table 17.1** Parallel texts from the Swiss federal law on milk transportation

German	French	Italian
Art. 35 Milchtransport	Art. 35 Transport du lait	Art. 35 Trasporto del latte
1. Die Milch ist schonend und hygienisch in den Verarbeitungsbetrieb zu transportieren. Das Transportfahrzeug ist stets sauber zu halten. Zusammen mit der Milch dürfen keine Tiere und milchfremde Gegenstände transportiert werden, welche die Qualität der Milch beeinträchtigen können.	1. Le lait doit être transporté jusqu'à l'entreprise de transformation avec ménagement et conformément aux normes d'hygiène. Le véhicule de transport doit être toujours propre. Il ne doit transporter avec le lait aucun animal ou objet susceptible d'en altérer la qualité.	1. Il latte va trasportato verso l'azienda di trasformazione in modo accurato e igienico. Il veicolo adibito al trasporto va mantenuto pulito. Con il latte non possono essere trasportati animali e oggetti estranei, che potrebbero pregiudicarne la qualità.
2. Wird Milch ausserhalb des Hofes zum Abtransport bereitgestellt, so ist sie zu beaufsichtigen.	2. Si le lait destiné à être transporté est déposé hors de la ferme, il doit être placé sous surveillance.	2. Se viene collocato fuori dall'azienda in vista del trasporto, il latte deve essere sorvegliato.
3. Milchpipelines sind nach den Anweisungen des Herstellers zu reinigen und zu unterhalten.	3. Les lactoducs des exploitations d'estivage doivent être nettoyés et entretenus conformément aux instructions du fabricant.	3. I lattodotti vanno puliti e sottoposti a manutenzione secondo le indicazioni del fabbricante.

excerpt of the Swiss federal law in German, French, and Italian on the quality of milk production.

The idea of machine translation with parallel texts is simple: given a sentence, a phrase, or a word in a **source language**, find its equivalent in the **target language**. The translation procedure splits the text to translate into fragments, finds a correspondence for each source fragment in the parallel corpora, and composes the resulting target pieces to form a translated text. Using the titles in Table 17.1, we can build pairs from the phrases *transport du lait* ‘milk transportation’ in French, *Milchtransport* in German, and *trasporto del latte* in Italian.

The idea of translating with the help of parallel texts is not new and has been applied by many people. A notable example is the Egyptologist and linguist Jean-François Champollion, who used the famous Rosetta Stone, an early parallel text, to decipher Egyptian hieroglyphs from Greek.

## 17.2 Alignment

The parallel texts must be aligned before using them in machine translation. This corresponds to a preliminary segmentation and mark-up that determines the corresponding paragraphs, sentences, phrases, and possibly words across the texts. Alignment of texts in Table 17.1 is made easier because paragraphs are numbered

and have the same number of sentences in each language. Some corpora, like Tatoeba,<sup>1</sup> have even pairs of translated sentences. This is not always the case, however, and some texts show a significantly different sentence structure.

Gale and Church (1993) describe a simple and effective method based on the idea that

longer sentences in one language tend to be translated into longer sentences in the other language, and that shorter sentences tend to be translated into shorter sentences.

Their method generates pairs of sentences from the target and source texts, assigns them a score, which corresponds to the difference of lengths in characters of the aligned pairs, and uses dynamic programming to find the maximum likelihood alignment of sentences.

The sentences in the source language are denoted  $s_i$ ,  $1 \leq i \leq I$ , and the sentences in the target language  $t_i$ ,  $1 \leq i \leq J$ .  $D(i, j)$  is the minimum distance between sentences  $s_1, s_2, \dots, s_i$  and  $t_1, t_2, \dots, t_j$ , and  $d(\text{source}_1, \text{target}_1; \text{source}_2, \text{target}_2)$  is the distance function between sentences. The algorithm identifies six possible cases of alignment through insertion, deletion, substitution, expansion, contraction, or merger. They are expressed by the formula below:

$$D(i, j) = \min \left( \begin{array}{l} D(i, j - 1) + d(0, t_j; 0, 0) \\ D(i - 1, j) + d(s_i, 0; 0, 0) \\ D(i - 1, j - 1) + d(s_i, t_j; 0, 0) \\ D(i - 1, j - 2) + d(s_i, t_j; 0, t_{j-1}) \\ D(i - 2, j - 1) + d(s_i, t_j; s_{i-1}, 0) \\ D(i - 2, j - 2) + d(s_i, t_j; s_{i-1}, t_{j-1}) \end{array} \right).$$

The distance function is defined as  $-\log P(\text{alignment}|\delta)$ , with

$$\delta = (l_2 - l_1 c) / \sqrt{l_1 s^2},$$

and where  $l_1$  and  $l_2$  are the lengths of the sentences under consideration,  $c$  the average number of characters in the source language  $L_2$  per character in the target language  $L_1$ , and  $s^2$  its variance. Gale and Church (1993) found a value of  $c$  of 1.06 for the pair French–English and 1.1 for German–English. This means that French and German texts are longer than their English counterparts: 6% longer for French and 10% for German. They found  $s^2 = 7.3$  for German–English and  $s^2 = 5.6$  for French–English.

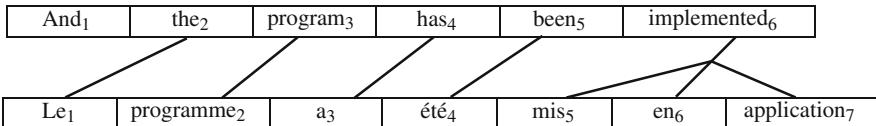
Using Bayes' theorem, we can derive a new distance function:

$$-\log P(\delta|\text{alignment}) - \log P(\text{alignment}).$$

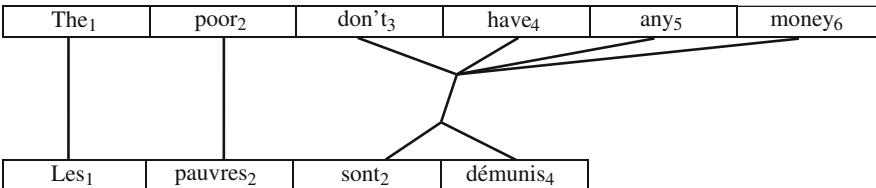
Gale and Church (1993) estimated the probability  $P(\text{alignment})$  of their six possible alignments with these figures: substitution 1–1: 0.89, deletion and substitution

---

<sup>1</sup> <https://tatoeba.org/>.



**Fig. 17.1** Alignment. After Brown et al. (1993)



**Fig. 17.2** A general alignment. After Brown et al. (1993)

0–1 or 1–0: 0.0099, expansion and contraction 2–1 or 1–2: 0.089, and merger 2–2: 0.011. They rewrote  $P(\delta|alignment)$  as  $2(1 - P(|\delta|))$ , which can be computed from statistical tables. See Gale and Church's original article.

Alignment of words and phrases uses similar techniques, however, it is more complex. Figures 17.1 and 17.2 show examples of alignment from Brown et al. (1993).

### 17.3 The Encoder-Decoder Architecture

In Chaps. 14, *Part-of-Speech and Sequence Annotation*, 15, *Self-Attention and Transformers*, and 16, *Pretraining an Encoder: The BERT Language Model*, we saw techniques to annotate a sequence of words with a sequence of tags: parts of speech, chunks, or named entities. So why not try to replace the tags with the words of a target language instead? Unfortunately, this idea will not fare very long as most pairs have different lengths.

The encoder-decoder architecture is a technique to translate a sequence into another sequence with no such a constraint on their lengths. We saw in Sect. 10.8 that, using a language model, it was possible to generate a sequence from one seed symbol, either a word or a character. Using an autoregressive technique, at step  $i$ , the generator outputs one symbol from the results of steps  $1, \dots, i - 1$ :

$$\text{generator}(y_1, y_2, \dots, y_{i-1}) = y_i.$$

The  $y_i$  symbol is appended to the input and the generation repeats until an end condition is satisfied. We also saw in Chap. 16 *Pretraining an Encoder: The BERT Language Model* that the encoder-transformers could build a semantic representation of their input.

The transformer consists of two similar components, an encoder like the one in Chap. 15 *Self-Attention and Transformers* and a decoder, assembled to carry out sequence-to-sequence transductions:

1. The encoder builds a representation of an input sequence,  $(x_1, x_2, \dots, x_n)$  using a language model. Vaswani et al. (2017) called this representation the memory,  $M$ ,

$$\text{encoder}(x_1, x_2, \dots, x_n) = M;$$

2. The decoder, equipped with a similar language model, uses  $M$  and a start symbol  $\langle s \rangle$  as first input to generate a target sequence. At each step, the decoder generates a new output symbol from the concatenation of the previous input and the last output. In addition to the tag to mark the start of the sequence, it uses a second one to tell it to stop,  $\langle /s \rangle$ :

$$\begin{aligned} \text{decoder}(M, \langle s \rangle) &= y'_1, \\ \text{decoder}(M, \langle s \rangle, y'_1) &= y'_2, \\ \text{decoder}(M, \langle s \rangle, y'_1, y'_2) &= y'_3, \\ &\dots \\ \text{decoder}(M, \langle s \rangle, y'_1, y'_2, \dots, y'_{p-1}) &= y'_p, \\ \text{decoder}(M, \langle s \rangle, y'_1, y'_2, \dots, y'_{p-1}, y'_p) &= \langle /s \rangle. \end{aligned}$$

For two source and target sequences of characters in English and French:

Source: ('H', 'e', 'l', 'l', 'o')  
 Target: ('B', 'o', 'n', 'j', 'o', 'u', 'r')

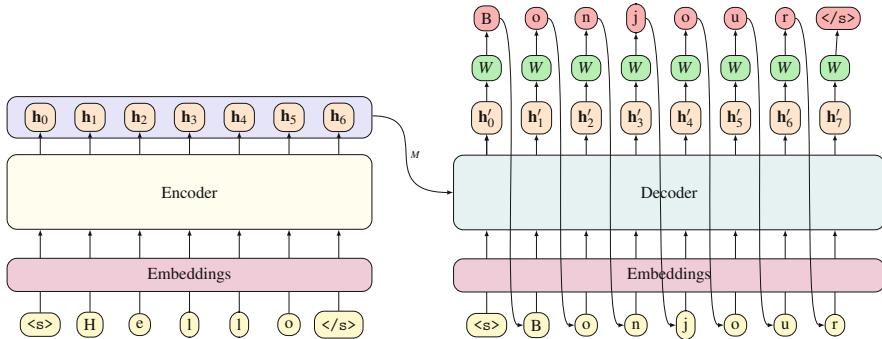
we would first encode *Hello*, the source sequence:

$$\text{encoder}(H, e, l, l, o) = M;$$

and, using  $M$  and starting with  $\langle s \rangle$ , iteratively decode the target sequence:

$$\begin{aligned} \text{decoder}(M, \langle s \rangle) &= B, \\ \text{decoder}(M, \langle s \rangle, B) &= o, \\ \text{decoder}(M, \langle s \rangle, B, o) &= n, \\ &\dots \\ \text{decoder}(M, \langle s \rangle, B, o, n, j, o, u) &= r, \\ \text{decoder}(M, \langle s \rangle, B, o, n, j, o, u, r) &= \langle /s \rangle, \end{aligned}$$

where  $\langle /s \rangle$  stops the generation. In fact, we simplified the decoder output as it is a vector. To be complete, we would need a linear layer to map it to the character prediction. Figure 17.3 summarizes this architecture.



**Fig. 17.3** Encoder-decoder architecture

## 17.4 Encoder-Decoder Transformers

In Chap. 15, *Self-Attention and Transformers*, we saw the encoder architecture with multihead attention and feed-forward networks and, in Chap. 16, *Pretraining an Encoder: The BERT Language Model*, how we can pretrain it as a large language model and use it in classification, sequence annotation, and question answering. Figure 17.4 shows now the complete transformer architecture, which consists of an encoder to the left and a decoder to the right.

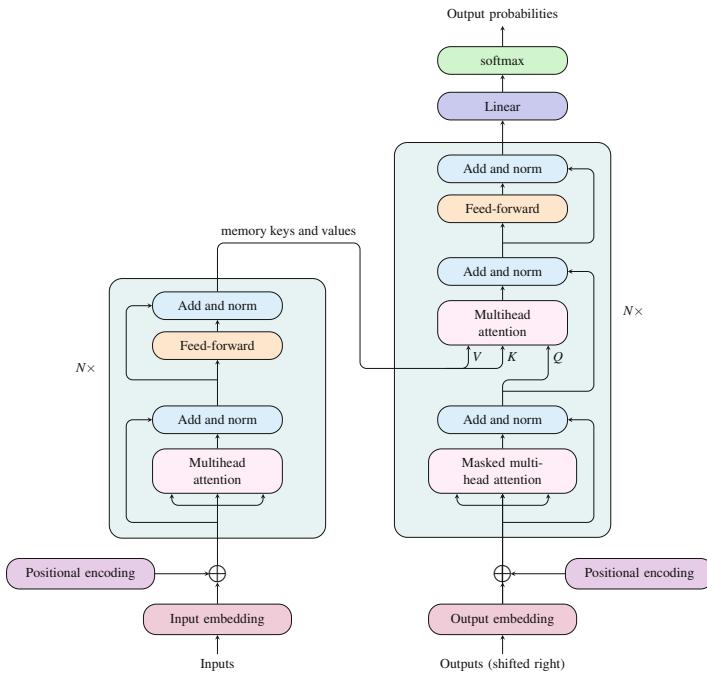
The decoder also builds on multihead attention, but with slight differences: Its first attention module uses a mask to make it auto-regressive and the second one merges the source and the target so that the sequence positions in the decoder can attend those in the encoder. We describe these differences now.

### 17.4.1 Masked Attention

As we said, Vaswani et al. (2017) first applied encoder-decoder transformers to translation. In Fig. 17.4, the inputs and outputs correspond to the source and target pair. The shifted outputs are simply the target sequences with a start symbol  $\langle s \rangle$  to initialize the decoding:

$$\begin{array}{ll} \textbf{Output :} & B \quad o \quad n \quad j \quad o \quad u \quad r \quad \langle /s \rangle \\ & \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\ \textbf{Input :} & \langle s \rangle \quad H \quad e \quad l \quad l \quad o \quad \langle /s \rangle \quad \textbf{Shifted output :} & \langle s \rangle \quad B \quad o \quad n \quad j \quad o \quad u \quad r \end{array}$$

When translating a sequence, the source is known in advance, while the target is decoded one symbol at a time, either word, subword, or character. This does not fit the format of the dataset when we train the model because the attention function, as we described it earlier, has a complete access to the source and target pairs. To have



**Fig. 17.4** The transformer consisting of  $N$  identical encoder layers (left) and  $N$  decoder layers (right). After Vaswani et al. (2017)

the same information as during prediction, at position  $i$  of the decoded sequence, the decoder stack should only pay attention to the symbols before it.

Vaswani et al. (2017) modified the attention layer in consequence and added a mask to indicate the positions to attend. Practically, we add a strictly upper triangular matrix,  $U_{-\infty}$ , filled with  $-\infty$  values, to  $QK^\top$  so that the attention weights beyond  $i$  are set to 0 by the softmax function:

$$\text{MaskedAttention}(Q, K, V, U_{-\infty}) = \text{softmax} \left( \frac{QK^\top}{\sqrt{d_k}} + U_{-\infty} \right) V.$$

Let us modify the `attention()` function from Sect. 15.2.5 to include this  $U_{-\infty}$  matrix. We first create a mask with a size parameter corresponding to the length of the sequence with the function:

```
def attn_mask(size):
    U = torch.empty(size, size).fill_(float('-inf'))
    return torch.triu(U, diagonal=1)
```

	i	must	go	back	to	my	ship	and	to	my	crew
i	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
must	0.42	0.58	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
go	0.44	0.22	0.35	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
back	0.29	0.11	0.19	0.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00
to	0.20	0.20	0.16	0.17	0.27	0.00	0.00	0.00	0.00	0.00	0.00
my	0.30	0.05	0.08	0.07	0.04	0.45	0.00	0.00	0.00	0.00	0.00
ship	0.04	0.04	0.04	0.05	0.06	0.04	0.73	0.00	0.00	0.00	0.00
and	0.14	0.10	0.09	0.13	0.16	0.12	0.05	0.21	0.00	0.00	0.00
to	0.12	0.12	0.10	0.11	0.16	0.10	0.04	0.08	0.16	0.00	0.00
my	0.20	0.03	0.05	0.05	0.03	0.29	0.01	0.02	0.03	0.29	0.00
crew	0.06	0.05	0.05	0.06	0.05	0.06	0.21	0.04	0.05	0.06	0.31

**Fig. 17.5** Masked self-attention weights. This table uses GloVe 50d vectors trained on Wikipedia 2014 and Gigaword 5. Compare with the results in Fig. 15.4

that fills a square matrix with  $-\infty$  values and sets the lower part to 0 with `torch.triu()`:

```
>>> attn_mask(5)
tensor([[0., -inf, -inf, -inf, -inf],
        [0., 0., -inf, -inf, -inf],
        [0., 0., 0., -inf, -inf],
        [0., 0., 0., 0., -inf],
        [0., 0., 0., 0., 0.]])
```

We then modify our attention function to add the mask to  $QK^\top$ :

```
def attention_masked(Q, K, V, U):
    d_k = K.size(dim=-1)
    attn_weights = F.softmax(Q @ K.T/math.sqrt(d_k)
                            + U, dim=-1)
    attn_output = attn_weights @ V
    return attn_output, attn_weights
```

We can now repeat the experiment with *I must go back to my ship and to my crew* from Chap. 15 *Self-Attention and Transformers* and compute the attention weights. We see that while the initial weights in Fig. 15.4 take into account the complete sentence, the masked ones in Fig. 17.5 consider the past words only. When multiplying the weights with the  $V$  matrix, the word *I* keeps its initial embedding vector; the embedding vector of *must* is the weighted sum of its original one and that of *I*; for *go*, it will be of itself, *I* and *must*, etc.

### 17.4.2 Cross-Attention

In the early days of neural machine translation, Bahdanau et al. (2015) showed that encoder-decoder architectures benefited from the decoder attending all the positions of an encoded sequence. This is precisely the purpose of the decoder’s second

attention layer in Fig. 17.4, which is fed with the memory, the encoded value of the input, and combined with the result of the decoder’s first attention. In this operation, the memory provides the key and the value,  $K_{\text{enc}}$  and  $V_{\text{enc}}$ , and the decoder the query,  $Q_{\text{dec}}$ :

$$\text{softmax} \left( \frac{Q_{\text{dec}} K_{\text{enc}}^T}{\sqrt{d_k}} \right) V_{\text{enc}}.$$

The operation is no longer a self-attention, but a cross-attention as it involves the source and target sequences.

In this second multihead attention operation, the matrix sizes are different and this might be puzzling. Let us detail the computation with a one head attention so that  $d_k$  is set to dimensionality of the embeddings,  $d_{\text{model}}$ . We have three matrices, where two come from the encoder and are identical,  $K_{\text{enc}} = V_{\text{enc}} = M$ , and one from the decoder,  $Q_{\text{dec}}$ . We check that their sizes are compatible with a matrix multiplication:

- A source input sequence of  $n$  tokens or symbols, for instance  $(H, e, l, l, o)$ , results in an encoder output  $M$  of size  $n \times d_{\text{model}}$ ;
- A target input sequence of  $p$  tokens, for instance  $(< \text{s} >, B, o, n, j, o, u, r)$ , results in a matrix of size  $p \times d_{\text{model}}$ . This is also the size of  $Q_{\text{enc}}$  after the masked attention layer.

Setting aside the three linear functions at the input of the cross-attention layer, we have:

$$\text{CrossAttention}(Q, M) = \text{softmax} \left( \frac{Q_{\text{dec}} M^T}{\sqrt{d_{\text{model}}}} \right) M.$$

The product  $Q_{\text{dec}} M^T$  has a size of  $p \times n$ , which is not modified by the softmax function, and, after multiply it with  $M$ , we have  $p \times d_{\text{model}}$ . This confirms that the lengths of the target input sequence of the decoder (bottom right of Fig. 17.4) and that of the output sequence of the transformer (top right of the figure) are identical:  $p$  tokens.

In addition to the connection created by cross-attention, the transformer shares the embeddings matrices between the input, output, and the last linear layer of the decoder.

### 17.4.3 Evaluating Translation

Vaswani et al. (2017) trained their transformer on English-French and English-German sentence pairs, respectively 36 millions and 4.5 millions. For both, they used a subword tokenization, either with WordPiece for French or BPE for German. This enabled them to limit the number of tokens to respectively 32,000 and 37,000.

As metric, they used the bilingual evaluation understudy (BLEU) algorithm (Papineni et al. 2002). BLEU compares the machine translation of a text with corresponding human translations. It uses a test set, where each sentence is translated by one or more human beings, and computes a score for each sentence and an average on the test set. The most basic score is a word-for-word comparison. It corresponds to the number of machine-translated words that appear in the human translations divided by the total number of words in the machine-translated sentence. The final score on the test set ranges from 0 to 1.

Papineni et al. noted that sequences of repeated words, such as articles, could reach high scores even if they made no sense. They modified their first algorithm in consequence by setting a maximal count for each word. This maximal count is computed from the human translations.

BLEU extends the word-for-word comparison to  $n$ -grams with the counts of machine-translated  $n$ -grams matching the human translations and divided by the total number of  $n$ -grams in the machine-translated sentence.

When they published their paper, Vaswani et al. (2017) reported BLEU scores of 41 for the French-English pair and of 26.4 for German-English. This was better than the previous state of the art at that time.

## 17.5 Programming: Machine Translation

We will now exemplify the transformer architecture with a machine translation program. We will use a corpus of aligned sentences in French and English from the Tatoeba project<sup>2</sup> in a curated version from the Anki site.<sup>3</sup> In addition, we will replace the subword input from Vaswani et al. (2017) with character sequences as it simplifies the programs.

As for encoders in Sect. 15.12, PyTorch has ready-to-use decoder components that follow the structure of Fig. 17.4, right part. It also has a complete transformer class that includes an encoder and a decoder. We first describe these classes and we will then use the transformer class to build the translation model.

### 17.5.1 The PyTorch TransformerDecoder Class

The programming scheme of a decoder is the same as with the encoder. We first create a decoder layer with `TransformerDecoderLayer()`:

```
decoder_layer = nn.TransformerDecoderLayer(d_model,
                                           nhead,
                                           batch_first=True)
```

---

<sup>2</sup> <https://tatoeba.org/>.

<sup>3</sup> <https://www.manythings.org/anki/>.

and then a decoder stack of  $N$  identical layers with `nn.TransformerDecoder`:

```
decoder = nn.TransformerDecoder(decoder_layer, num_layers)
```

Once created, the input to a decoder object is a target and a memory. As optional arguments, we usually add an upper triangular target mask, `tgt_mask`, to have an auto-regressive model and two padding masks for the target and the memory, respectively `tgt_key_padding_mask` and `memory_key_padding_mask`. They remove the padding tokens of the mini-batches from the attention mechanism as we did with the encoder in Sect. 15.12.

```
dec_output = decoder(
    target,
    memory,
    tgt_mask=U,
    tgt_key_padding_mask=tgt_padding,
    memory_key_padding_mask=mem_padding)
```

## 17.5.2 The PyTorch Transformer Class

The `nn.Transformer` class is even easier to use than the decoder. The statement

```
transformer = nn.Transformer(batch_first=True)
```

creates a transformer with default arguments that are the same as in the base model of Vaswani et al.:

- `d_model = 512`;
- `nhead = 8`
- `num_encoder_layers` and `num_decoder_layers` set to 6 and;
- `dim_feedforward = 2048`.

See Table 3 in their paper.

Inside the `nn.Transformer` class, the `__init__()` method creates an encoder and a decoder. Its `forward()` method applies them to a source, a target, and a target mask:

```
trans_output = transformer(source,
                           target,
                           tgt_mask=U)
```

where the encoder encodes the source and returns a memory and the decoder decodes this memory and the target. We access the encoder and decoder objects with:

```
>>> transformer.encoder
TransformerEncoder(
(layers): ModuleList(
(0-5): 6 x TransformerEncoderLayer(
  (self_attn): MultiheadAttention(
    ...
    ...
  )
)
)
)
```

and

```
>>> transformer.decoder
TransformerDecoder(
    layers): ModuleList(
        (0-5): 6 x TransformerDecoderLayer(
            self_attn): MultiheadAttention(
                ...

```

The transformer has padding mask arguments as with the encoder or decoder alone. Additionally, `nn.Transformer` has a static method to create an upper triangular mask that we denoted  $U$  in the previous sections:

```
nn.Transformer.generate_square_subsequent_mask(size)
```

### 17.5.3 Preparing the Dataset

A Tatoeba bilingual corpus<sup>4</sup> consists of pairs of sentence, each preceded by a sentence identifier as for instance:

16492 What are you doing? 4306545 Qu'est-ce que vous faites ?

We download the Anki version of it:<sup>5</sup> A subset limited to sentences written by native speakers. Its format is slightly different and contains the attribution of the text:

What are you doing? Qu'est-ce que vous faites ? CC-BY 2.0 (France) Attribution: tatoeba.org #16492 (CK) & #4306545 (gillux)

We read the sentences:

```
with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().strip().split('\n')
```

And we count them:

```
>>> len(lines)
229803
```

This is already quite a large number and we limit it to 50,000 pairs. We create two lists of source and target sentences with less than 100 characters:

```
max_len = 100
num_samples = 50000

def create_pairs(lines, max_len, num_samples, shuffle=True):
    src_seqs = []
    tgt_seqs = []
    if shuffle:
```

---

<sup>4</sup> <https://tatoeba.org/downloads>.

<sup>5</sup> <https://www.manythings.org/anki/fra-eng.zip>.

```

        random.shuffle(lines)
    for line in lines:
        src_seq, tgt_seq, _ = line.split('\t')
        if len(src_seq) <= max_len and len(tgt_seq) <= max_len:
            src_seqs.append(src_seq)
            tgt_seqs.append(tgt_seq)
    return src_seqs[:num_samples], tgt_seqs[:num_samples]

tgt_seqs, src_seqs = create_pairs(lines, max_len, num_samples)

```

French will be source language and English the target one. We have:

```

>>> src_seqs[5]
'Je suis sans argent.'
>>> tgt_seqs[5]
"I haven't got any money."

```

We can now split our dataset into training and validation sets with the proportion 80/20. We compute the train/validation index:

```

TRAIN_PERCENTAGE = 0.8
train_val = int(TRAIN_PERCENTAGE * num_samples)

```

and we create the sets:

```

train_src_seqs = src_seqs[:train_val]
train_tgt_seqs = tgt_seqs[:train_val]

val_src_seqs = src_seqs[train_val:]
val_tgt_seqs = tgt_seqs[train_val:]

```

This results into 40,000 training samples and 10,000 validation ones.

#### 17.5.4 Indexing the Symbols

So far, our data consists of strings of characters. We will now convert them into lists of indices. This is a procedure we have already done a couple of times. The language pair will share the same vocabulary as in Vaswani et al. (2017).

We first collect the characters from the source and target training sets and we extract the character set:

```

src_chars = set(''.join(train_src_seqs))
tgt_chars = set(''.join(train_tgt_seqs))
charset = sorted(
    list(set.union(src_chars,
                  tgt_chars)))

```

We reserve four special tokens for the padding and unknown symbols as well as for the beginning and end of sequences:

```
special_tokens = ['<pad>', '<unk>', '<s>', '</s>']
```

We add them to the character set and we build index-to-token and token-to-index dictionaries:

```
charset = special_tokens + charset
idx2token = dict(enumerate(charset))
token2idx = {char: idx for idx, char in idx2token.items()}
```

We have now:

```
>>> idx2token
{0: '<pad>', 1: '<unk>', 2: '<s>', 3: '</s>', 4: ' ',
 5: '!', 6: '"', 7: '$', ...}
```

In total, 110 symbols.

We have now the data structures to convert the list of character strings into a list of index tensors:

```
def seqs2tensors(seqs, token2idx):
    tensors = []
    for seq in seqs:
        seq = ['<s>'] + list(seq) + ['</s>']
        tensors += torch.tensor([
            list(map(lambda x: token2idx.get(x, 1),
                     seq))]) # <unk> -> 1
    return tensors
```

and the reverse:

```
def tensors2seqs(tensors, idx2token):
    seqs = []
    for tensor in tensors:
        tensor = list(tensor)
        seqs += [list(
            map(lambda x: idx2token.get(x.item(), '<unk>'),
                tensor))]
    return seqs
```

We have:

```
>>> train_tgt_seqs[:2]
[ "I've always been very proud of you.",
  "I'm aware of the consequences." ]
```

and

```
>>> seqs2tensors(train_tgt_seqs[:2], token2idx)
[tensor([ 2, 36,  9, 75, 58,  4, 54, ...,  3]),
 tensor([ 2, 36,  9, 66,  4, 54, 76, ...,  3])]
```

### 17.5.5 Building the Transformer Model

The data input in Fig. 17.4 starts with the embedding of the source and target sequences. We already implemented an `Embedding` class in Sect. 15.10 that we reuse

here. The transformer itself is the PyTorch class from the previous section and the last layer is just a linear module. We compose a model from these components so that it encapsulates all the parameters we have to fit.

- In the `__init__()` method, we create the embedding layers, the transformer itself, and the last linear layer. As in Vaswani et al. (2017), we share the embedding weights;
- In the `__forward__()` method, we embed the source and the target sequences; we create the target autoregressive mask with the built-in transformer method; and we pass them to the transformer. We also pass the padding masks.

We use the same defaults as the `nn.Transformer()` class. Most of the code lines are parameter initializations:

```
class Translator(nn.Module):
    def __init__(self,
                 d_model=512,
                 nhead=8,
                 num_encoder_layers=6,
                 num_decoder_layers=6,
                 dim_feedforward=2048,  # 4 x d_model
                 dropout=0.1,
                 vocab_size=30000,
                 max_len=128):
        super().__init__()
        self.embedding = Embedding(vocab_size,
                                   d_model,
                                   max_len=max_len)
        self.transformer = nn.Transformer(
            d_model,
            nhead,
            num_encoder_layers,
            num_decoder_layers,
            dim_feedforward,
            dropout,
            batch_first=True)
        self.fc = nn.Linear(d_model, vocab_size)
        self.fc.weight = \
            self.embedding.input_embedding.weight

    def forward(self, src, tgt,
               src_padding, tgt_padding):
        src_embs = self.embedding(src)
        tgt_embs = self.embedding(tgt)
        tgt_mask = \
            nn.Transformer.generate_square_subsequent_mask(
                tgt.size(dim=1))
        x = self.transformer(
            src_embs,
            tgt_embs,
            tgt_mask=tgt_mask,
            src_key_padding_mask=src_padding,
            memory_key_padding_mask=src_padding,
```

```
tgt_key_padding_mask=tgt_padding)
return self.fc(x)
```

With this class, we create a translator object:

```
translator = Translator(vocab_size=len(idx2token))
```

We can now apply our transformer to an input and examine its output. We first format the source and target characters, respectively: We select a mini-batch of 32 samples, convert the two lists of character strings to two lists of tensors, and pad them to obtain tensors:

```
src_batch = pad_sequence(seqs2tensors(
    train_src_seqs[:32], token2idx),
    batch_first=True, padding_value=0)
tgt_batch = pad_sequence(seqs2tensors(
    train_tgt_seqs[:32], token2idx),
    batch_first=True, padding_value=0)
```

We have now two tensors representing the source and the target.

We create the padding masks so that we remove the padding symbols from the attention:

```
src_padding_mask = (src_batch == 0)
tgt_padding_mask = (tgt_batch == 0).float()
```

We finally apply the translator object to the mini-batch:

```
>>> translator.eval()
>>> translator(src_batch, tgt_batch,
    src_padding_mask, tgt_padding_mask).size()
torch.Size([32, 64, 110])
```

as result, we have 32 samples of 64-character long, the maximal length of our mini-batch here, and logits for 110 characters, the size of our character set.

We can even extract the predictions by taking the maximal values on the last axis and map them to characters with

```
>>> tensors2seqs(
    torch.argmax(
        translator(src_batch, tgt_batch,
            src_padding_mask, tgt_padding_mask),
        dim=-1),
    idx2token)
```

```
[[H', m', m', 2', m', E', ...], ...]].
```

This is of course nonsensical as the matrices still have random values.

## 17.5.6 Configuring the Model and Training Procedure

Training a model without an expensive GPU can take a lot of time. Here we will use parameters smaller than will not require such an extension for our limited

experiment. We set  $d_{\text{model}}$  to 512,  $d_{\text{ff}}$  to 512, and the number of layers to 3. We add 2 to the sequence maximal length as we will pad them with  $\langle \text{s} \rangle$  and  $\langle / \text{s} \rangle$  symbols:

```
translator = Translator(d_model=512,
                       nhead=8,
                       num_decoder_layers=3,
                       num_encoder_layers=3,
                       dim_feedforward=512,
                       vocab_size=len(token2idx),
                       max_len=max_len + 2)
```

In his pedagogical implementation of an *Annotated Transformer*, Rush (2018) noted that the parameter initialization was essential and not documented in the original paper. This has significant consequences on the convergence. He proposed an initialization that we follow here:

```
for p in translator.parameters():
    if p.requires_grad and p.dim() > 1:
        nn.init.xavier_uniform_(p)
```

Vaswani et al. (2017) used the Adam optimizer with a variable learning rate. We simplify it with a constant one instead, the rest of the parameters is the same:

```
optimizer = torch.optim.Adam(
    translator.parameters(), lr=0.0001, betas=(0.9, 0.98),
    eps=1e-9)
```

We also simplify their loss and use a cross entropy:

```
loss_fn = torch.nn.CrossEntropyLoss(ignore_index=0)
```

### 17.5.7 Datasets and Dataloaders

In the previous chapters, we stored our datasets with the built-in `TensorDataset`, a subclass of `Dataset`, the generic representation of a dataset in PyTorch. Here our input consists of pairs of tensors. Such a class does not exist and we have to create it. To derive a class from `Dataset`, according to the PyTorch documentation, we must implement three methods: `__init__`, `__len__`, and `__getitem__`, that respectively creates the dataset object, returns its length, and extracts an item from it.

In `__init__`, we pass the dataset in the form of lists of source and target sequences; in `__len__()`, we compute the size of the dataset, and in `__getitem__()`, we return a pair at a certain index. To create a batch, the dataloader uses `__getitem__` as many times as we have samples in the batch. As the sequences have different lengths, we add a `collate` function to convert the source and target batches in two tensors.

```
class PairDataset(Dataset):
    def __init__(self, src_seqs, tgt_seqs,
                 token2idx):
        self.src_seqs = src_seqs
```

```

    self.tgt_seqs = tgt_seqs
    self.token2idx = token2idx

    def __len__(self):
        return len(self.src_seqs)

    def __getitem__(self, idx):
        src_batch = seqs2tensors([self.src_seqs[idx]],
                                self.token2idx)
        tgt_batch = seqs2tensors([self.tgt_seqs[idx]],
                                self.token2idx)
        return src_batch[0], tgt_batch[0]

    def collate(self, batch):
        src_batch, tgt_batch = list(zip(*batch))
        src_batch = pad_sequence(src_batch, batch_first=True,
                                padding_value=0)
        tgt_batch = pad_sequence(tgt_batch, batch_first=True,
                                padding_value=0)
        return src_batch, tgt_batch

```

With this class, we create a `PairDataset` object to store the pairs .

```
train_dataset = PairDataset(train_src_seqs, train_tgt_seqs,
                            token2idx)
```

and we pass it to the dataloader that will carry out the iteration:

```
train_dataloader = DataLoader(train_dataset, batch_size=32,
                            shuffle=True,
                            collate_fn=train_dataset.collate)
```

The `collate_fn` argument indicates the function to collate the samples into a batch.

We create similar objects for the validation dataset: `val_dataset` and `val_dataloader`.

### 17.5.8 Training the Model

We can now train the model. We first write an evaluation function as it is a bit easier. We build the target output by removing the first character, `<s>` and the target input by removing the last one so that the target sequences have the same length. We create the padding mask and we apply the model. The computation of the loss is similar to that in Sect. 14.8.1.

```

def evaluate(model, loss_fn, dataloader):
    model.eval()
    with torch.no_grad():
        t_loss = 0
        t_correct, t_chars = 0, 0
        for src_batch, tgt_batch in dataloader:
            tgt_input = tgt_batch[:, :-1]

```

```

tgt_output = tgt_batch[:, 1:]
src_padding_mask = (src_batch == 0)
tgt_padding_mask = (tgt_input == 0).float()
tgt_output_pred = model(src_batch, tgt_input,
                        src_padding_mask,
                        tgt_padding_mask)
loss = loss_fn(
    tgt_output_pred.reshape(
        -1,
        tgt_output_pred.size(dim=-1)),
    tgt_output.reshape(-1))
n_chars = (tgt_output != 0).sum()
t_chars += n_chars
t_loss += loss.item() * n_chars
char_pred = torch.argmax(tgt_output_pred, dim=-1)
char_correct = torch.mul((char_pred == tgt_output),
                        (tgt_output != 0)).sum()
t_correct += char_correct.item()
return t_loss / t_chars, t_correct / t_chars

```

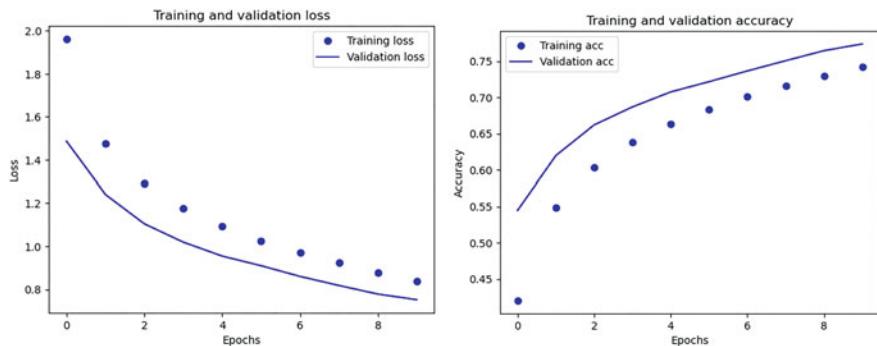
The training function adds the gradient computation and update to the previous function.

```

def train(model, loss_fn, optimizer, dataloader):
    model.train()
    t_loss = 0
    t_correct, t_chars = 0, 0
    for src_batch, tgt_batch in tqdm(dataloader):
        tgt_input = tgt_batch[:, :-1]
        tgt_output = tgt_batch[:, 1:]
        src_padding_mask = (src_batch == 0)
        tgt_padding_mask = (tgt_input == 0).float()
        tgt_output_pred = model(src_batch, tgt_input,
                               src_padding_mask,
                               tgt_padding_mask)
        optimizer.zero_grad()
        loss = loss_fn(
            tgt_output_pred.reshape(
                -1,
                tgt_output_pred.size(dim=-1)),
            tgt_output.reshape(-1))
        loss.backward()
        optimizer.step()
        with torch.no_grad():
            n_chars = (tgt_output != 0).sum()
            t_chars += n_chars
            t_loss += loss.item() * n_chars
            char_pred = torch.argmax(tgt_output_pred, dim=-1)
            char_correct = torch.mul((char_pred == tgt_output),
                                    (tgt_output != 0)).sum()
            t_correct += char_correct.item()

    return t_loss / t_chars, t_correct / t_chars

```



**Fig. 17.6** Training curves of the transformer with a Tatoeba French-to-English corpus. Loss and accuracy over 10 epochs

We run the training 10 epochs with the loop:

```
history = []
history['accuracy'] = []
history['loss'] = []
history['val_accuracy'] = []
history['val_loss'] = []

for epoch in range(EPOCHS):
    train_loss, train_acc = train(
        translator, loss_fn, optimizer, train_dataloader)
    history['loss'] += [train_loss]
    history['accuracy'] += [train_acc]
    val_loss, val_acc = evaluate(translator, loss_fn, val_dataloader)
    history['val_loss'] += [val_loss]
    history['val_accuracy'] += [val_acc]
```

Figure 17.6 shows the training curves across the epochs. They show that the model is still steadily improving and more epochs would be needed to fit it completely.

### 17.5.9 Decoding a Sentence

Now that we have trained a model, we can translate a sequence. We first embed the source characters and encode them into a `memory`. Then, starting from `<s>`, we embed the current target characters and decode them with the memory. We predict the next character with a linear layer that we apply to the last encoded vector. We select the highest probability. As the decoder is auto-regressive, we form the next input by the concatenation of the previous one and of the last output.

We repeat this operation always selecting the highest probability for the last character. This is a greedy decoding similar to that proposed by Rush (2018). We

terminate when we encounter an end of sequence, `</s>`, or, as in Vaswani et al. (2017), when the output is more than 50 character longer than the input.

```
def greedy_decode(model, src_seq, max_len):
    src_embs = model.embedding(src_seq)
    memory = model.transformer.encoder(src_embs)
    tgt_seq = torch.LongTensor([2]) # <s>
    tgt_embs = model.embedding(tgt_seq)
    for i in range(max_len-1):
        tgt_mask = nn.Transformer.generate_square_subsequent_mask(
            tgt_embs.size(dim=0))
        tgt_output = model.transformer.decoder(tgt_embs,
                                              memory,
                                              tgt_mask)
        char_prob = model.fc(tgt_output[-1])
        next_char = torch.argmax(char_prob)
        tgt_seq = torch.cat(
            (tgt_seq,
             torch.unsqueeze(next_char, dim=0)), dim=0)
        tgt_embs = model.embedding(tgt_seq)
        if next_char.item() == 3: # </s>
            break
    return tgt_seq[1:]
```

The top-level translation function is just a conversion of the source string into a tensor and a call the decoder, where we set a maximal length to the target. We clean the resulting list from the end-of-sequence token and convert it to a string:

```
def translate(model, src_sentence):
    model.eval()
    src = seqs2tensors([src_sentence], token2idx)[0]
    num_chars = src.size(dim=0)
    tgt_chars = greedy_decode(
        model, src, max_len=num_chars + 20)
    tgt_chars = tensors2seqs([tgt_chars], idx2token)[0]
    if tgt_chars[-1] == '</s>':
        tgt_chars = tgt_chars[:-1]
    tgt_str = ''.join(tgt_chars)
    return tgt_str
```

We can now run the translator on a few examples. Although far from perfect and sometimes really wrong, the results are quite promising for such a small model using characters only:

```
>>> translate(translator, 'Bonjour !')
'Good day.'
>>> translate(translator, "Va-t'en !")
'Go away.'
>>> translate(translator, 'Attends-moi !')
'Wait me.'
>>> translate(translator, 'Viens à la maison ce soir')
'Come home at home tonight.'
>>> translate(translator, 'Viens manger !')
'Come any money.'
```

### 17.5.10 Improving the Performance

The program we have written is merely a starting point and we can improve its performance with a few modifications. The best way is to follow Vaswani et al. (2017) and the didactical implementation of Rush (2018) in *The Annotated Transformer* notebook.<sup>6</sup> We outline a few options here:

1. First we can increase the amount of training data. The datasets from the Workshops on Machine Translation, for instance WMT 2014, are easy to obtain and among of the most popular ones in this field. They also serve as benchmark in the paper;<sup>7</sup>
2. Then, we should replace the character sequences with subwords. We can use the BPE algorithm to tokenize the texts, see Chap. 13, *Subword Segmentation*, and the SentencePiece program.<sup>8</sup> Vaswani et al. used 32,000 subwords for the English-French pair;
3. Vaswani et al.’s optimizer used a variable learning rate, where they started with vary low values, increased the rate linearly with the number of steps in the so-called warmup phase, and then decreased it with the number of steps;
4. Vaswani et al. also obtained an improvement by smoothing the output symbols, characters or subwords. When we compute a classic cross entropy, we encode the truth with unit vectors (one-hot encoding). Instead, for a given prediction, smoothing scales down the true unit vector,  $\mathbf{e}_i$ , and assigns the remaining probability mass to the other symbols  $\mathbf{e}_{j \neq i}$ . The new true distribution is:

$$(1 - \epsilon)\mathbf{e}_i + \frac{\epsilon}{N_{\text{subwords}} - 1} \sum_{j=1, j \neq i}^{N_{\text{subwords}}} \mathbf{e}_j,$$

where  $\epsilon = 0.1$ . We measure the loss between true and predicted distributions with the KL-divergence of Sect. 6.1.4. We can use the PyTorch `nn.KLDivLoss()` to implement it. We can also set the `label_smoothing` parameter of the `nn.CrossEntropyLoss()` class to 0.1;

5. In our implementation, we used a greedy decoding. Vaswani et al. (2017) used a beam search instead, where they decoded four concurrent sequences. At each decoding step, for each of the four sequences ( $< \mathbf{s} >, x_1^b, x_2^b, \dots, x_i^b$ ) with  $b$  ranging from one to four, beam search considers the four next tokens  $x_{i+1}^b$  with the highest scores. In total, we have 16 candidates for the four sequences. The

---

<sup>6</sup> <https://nlp.seas.harvard.edu/annotated-transformer/>.

<sup>7</sup> <https://huggingface.co/datasets/wmt14>.

<sup>8</sup> <https://github.com/google/sentencepiece>.

algorithm computes the resulting sequence probabilities for each candidate:

$$P(x_{i+1}^b | <\text{s}>, x_1^b, x_2^b, \dots, x_i^b) \prod_{j=1}^i P(x_j^b | <\text{s}>, x_1^b, x_2^b, \dots, x_{j-1}^b),$$

ranks the 16 sequences, and proceeds with the four best ones.

6. All this will require more computing capacity than that available on a common laptop. GPUs yield a significant acceleration. The PyTorch documentation provides tutorials on how to adapt a program to GPU computing.<sup>9</sup>

## 17.6 Decoders

After the encoder in Chaps. 15 *Self-Attention and Transformers* and 16, *Pretraining an Encoder: The BERT Language Model*, the encoder-decoder in this chapter, the standalone decoder is a third transformer architecture. In fact, such a decoder is an encoder with a masked input so that we can train it and run it with an autoregressive procedure, see Fig. 17.7, left.

The rationale behind it is that, if the encoder can build a semantic representation of the input, the decoder can do it as well as it has nearly the same structure. So why not feed the encoder with the input directly and then let it predict the output in an autoregressive way. Radford et al. (2018) expressed this as the maximization of the product of each prediction:

$$\prod_{i=1}^N P(x_i | x_1, x_2, \dots, x_{i-1}).$$

As example, Radford et al. (2019) proposed to train the model with sequences such as these ones:

```
answer the question, document, question, answer
translate to french, english text, french text
```

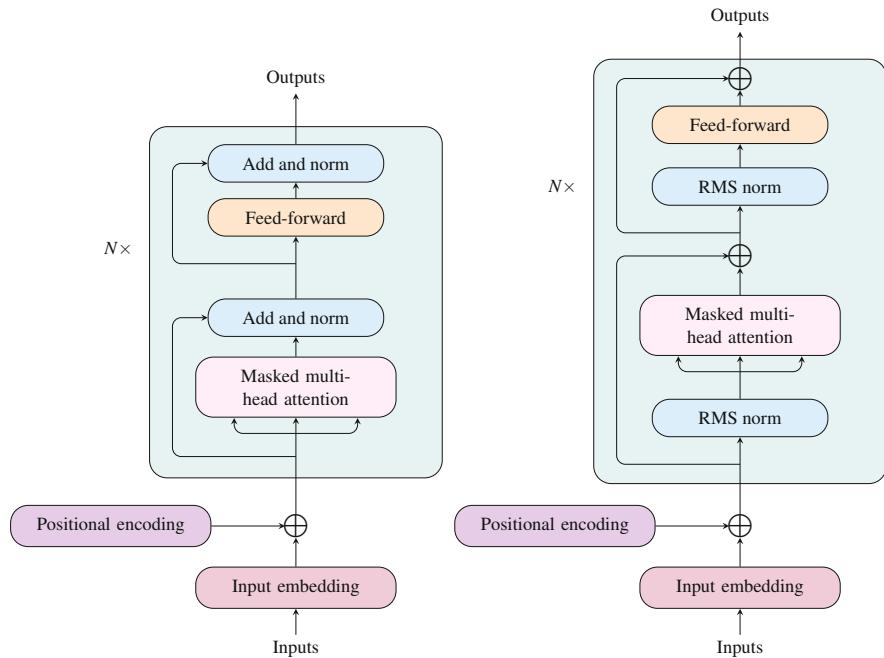
to answer questions and translate sentences. In both tasks, we help the decoder with a specification of what we expect it to do: answer a question or translate to French. In the rest, the data are similar to that in the BERT application in Sect. 16.8.3 or in translation in Sect. 17.4. Then given a task like

```
answer the question, document, question
translate to french, english text
```

the encoder would generate autoregressively either the `answer` or the `french text`.

---

<sup>9</sup> See for instance [https://pytorch.org/tutorials/beginner/translation\\_transformer.html](https://pytorch.org/tutorials/beginner/translation_transformer.html) on machine translation.



**Fig. 17.7** The transformer decoder with the post layer normalization, left, and prenormalization, right

More formally, we predict an output from an input and a task:

$$P(\text{output}|\text{input}, \text{task}).$$

The condition is usually called the **prompt** and the output, the **answer** or the **completion**.

### 17.6.1 Decoder Architecture

Recent decoder architectures, as implemented by Touvron et al. (2023a), have some peculiarities.

1. First, the normalization is applied before being passed to the attention and feed-forward sublayers, (see Fig. 17.7, right part):

$$X' = X + \text{MultiheadAttention}(\text{Norm}(X)).$$

In Sect. 15.4, we applied the normalization after.

2. Then, the Norm function is a unit normalization of the embedding vectors scaled by the embedding dimension,  $\sqrt{d_{\text{model}}} \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|}$  (Zhang and Sennrich 2019):

$$\begin{aligned}\text{RMSNorm}(x_{i,1}, x_{i,2}, \dots, x_{i,d_{\text{model}}}) &= \sqrt{\frac{d_{\text{model}}}{\sum_{j=1}^{d_{\text{model}}} x_{i,j}^2}} \cdot (x_{i,1}, x_{i,2}, \dots, x_{i,d_{\text{model}}}), \\ &= \frac{\sqrt{d_{\text{model}}}}{\|\mathbf{x}_i\|} \cdot \mathbf{x}_i.\end{aligned}$$

This normalization is faster than that in Sect. 15.4, as the input is simply rescaled.

3. All the ReLU activation functions,  $\max(0, x)$ , are replaced with a SwiGLU function (Shazeer 2020). This function uses the Swish function defined as:

$$\text{Swish}_{\beta}(x) = x\sigma(\beta x),$$

where  $\sigma$  is the logistic function. The activation applies Swish to a linear transformation of the input,  $\mathbf{xW} + \mathbf{b}$ , and computes the Hadamard product with another linear transformation,  $\mathbf{xV} + \mathbf{c}$ :

$$\text{Swish}_{\beta}(\mathbf{xW} + \mathbf{b}) \odot (\mathbf{xV} + \mathbf{c}).$$

ReLU sets all the negative numbers to zero and is nonlinear. SwiGLU is continuous and will not zero small negative values. Shazeer (2020) showed transformers obtained better results on the GLUE benchmark with the latter function.

4. Touvron et al. (2023a) replaced the absolute positional encoding with rotary positional embeddings (RoPE) (Su et al. 2024). Using a band matrix, we obtain them by applying rotations to pairs of coordinates in the initial position encoding. This technique enables the attention mechanism to inform two tokens of their relative positions.

In addition to these features, encoders contain scores of technical details that accumulated can make a major difference. We refer to Touvron et al. (2023a), Touvron et al. (2023b), and Jiang et al. (2023) for some recent ones.

## 17.6.2 Training Procedure

As with the BERT encoder, decoders are first pretrained on raw corpora. These corpora have now considerable sizes and represent significant portions of the textual web including Wikipedia and CommonCrawl,<sup>10</sup> scientific papers from arXiv,<sup>11</sup> and programming language code from GitHub. The internet contains many duplicates

---

<sup>10</sup> <https://commoncrawl.org/>.

<sup>11</sup> <https://arxiv.org/>.

and low-quality content. Wenzek et al. (2020) describe a technique to identify them using hashcodes and a language model. When deduplicated and cleaned, some pretraining corpora now surpass 2 trillion tokens.

So far, we did not specify the architecture parameters of an encoder and their number. As general principle, the encoder size must match that of the quantity of textual information. Current models, like Llama, have stacks of 32 to 80 encoding layers, a  $d_{\text{model}}$  value ranging from 4096 to 8192, 32 to 64 attention heads, a number of model parameters going from 7 to 70 billions. Once the model is created, the pretraining procedure to fit its parameters follows that of a supervised autoregressive language model and predicts the next token from the previous ones.

Once we have a pretrained model, we must adapt it to the different tasks we want it to complete. A first possibility is to fine-tune it on a set of examples as we saw with BERT (Sect. 16.8). The fine-tuning dataset consists of pairs of prompts and their completion. Given a pair as input, the model is finetuned with a supervised autoregressive technique just as in the pretraining step. The model parameters are adjusted so that they predict the correct completion. As public dataset of instruction pairs, Longpre et al. (2023) describe the Flan Collection of more than 15 million examples. This dataset is multilingual and the authors divided the tasks into 1800 different categories such as translate, summarize, or explain.

Decoder models are autoregressive. This means that, given an input, whatever it is, they start generating a sequence as in Sect. 10.8. An astonishing property is that they often output reasonable answers simply from their pretrained parameters. In addition to fine-tuning, a second technique is then to use this property called zero-shot. This can be complemented with one example, one-shot, or a few examples, few-shots, to guide even more the answer in a process called prompt engineering. For all these settings, the instruction, the examples, and the prompt are given as input and the model generates an answer. Brown et al. (2020) gives examples on translation that we reproduce in Fig. 17.8.

Why this works so well is not completely understood; it is probably due to the mass of knowledge stored in the matrices. Nonetheless, prompt engineering is not

**Fig. 17.8** Few-shot learning.

After Brown et al. (2020)

Zero-shot	
<i>Task description</i>	Translate English to French:
<i>Prompt</i>	cheese ⇒
One-shot	
<i>Task description</i>	Translate English to French:
<i>Example</i>	sea otter ⇒ loutre de mer
<i>Prompt</i>	cheese ⇒
Few-shot	
<i>Task description</i>	Translate English to French:
<i>Example</i>	sea otter ⇒ loutre de mer
<i>Example</i>	peppermint ⇒ menthe poivrée
<i>Example</i>	plush giraffe ⇒ girafe en peluche
<i>Prompt</i>	cheese ⇒

always trivial and may require a few iterations before the user gets what s/he wants from the machine.

## 17.7 Further Developments

In this last chapter, we have covered the transformer and transformer decoder architectures. These systems enabled natural language processing to make substantial progress in machine translation, question answering, or text generation. What makes them appealing is that they can scale in size with no or few architectural changes and digest a massive amount of knowledge. As a result, this started a race to gigantism, when large companies began to train transformers on colossal corpora, requiring huge computing resources. This development is unfortunately beyond the reach of most of us so far.

Fortunately, many authors of models now publish their parameters as well as their training programs. This spurs an incredible activity and creativity in the field, as anyone can reuse the models, sometimes in unexpected applications, fine-tune the parameters, optimize their representation, and even find ways to re-create them from scratch at a much lower cost. New models and benchmarking tasks appear at a stunning pace and, in a such rapidly developing field, it is difficult to write a definitive conclusion. The future is not written yet and the best we can do, to avoid obsolescence, is to get involved and try to shape it.

# 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., et al. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, Berkeley* (pp. 265–283). USENIX Association.
- Abeillé, A., & Clément, L. (2003). Annotation morpho-syntaxique. Les mots simples – Les mots composés. Corpus Le Monde. Technical report, LLF, Université Paris 7, Paris.
- Abeillé, A., Clément, L., & Tousseenel, F. (2003). Building a treebank for French. In A. Abeillé (Ed.), *Treebanks: Building and using parsed corpora. Text, speech and language technology* (chap. 10, vol. 20, pp. 165–187). Kluwer Academic.
- Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., & Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In J. Holub & J. Žd’árek (Eds.), *Implementation and Application of Automata. 12th International Conference on Implementation and Application of Automata, CIAA 2007, Prague, July 2007. Revised Selected Papers. Lecture notes in computer science* (vol. 4783, pp. 11–23). Springer.
- Antworth, E. L. (1994). Morphological parsing with a unification-based word grammar. In *North Texas Natural Language Processing Workshop*, University of Texas at Arlington.
- Antworth, E. L. (1995). *User’s guide to PC-KIMMO, Version 2*. Summer Institute of Linguistics, Dallas.
- Apache OpenNLP Development Community. (2012). *Apache OpenNLP developer documentation*. The Apache Software Foundation, 1.5.2 edition.
- Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. <http://arxiv.org/abs/1607.06450>.
- Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern information retrieval: The concepts and technology behind search* (2nd edn.). Addison-Wesley.
- Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In Y. Bengio & Y. LeCun (Eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings*.
- Beesley, K. R., & Karttunen, L. (2003). *Finite state morphology*. CSLI Publications.
- Beltrami, E. (1873). Sulle funzioni bilineari. *Giornale di Matematiche*, 11, 98–106.
- Bentivogli, L., Clark, P., Dagan, I., & Giampiccolo, D. (2009). The sixth pascal recognizing textual entailment challenge. In *Text Analysis Conference*.
- Bentley, J., Knuth, D., & McIlroy, D. (1986). Programming pearls. *Communications of the ACM*, 6(29), 471–483.

- Benzécri, F., & Morfin, M. (1981). *Introduction à l'analyse des correspondances et à la classification automatique* (vol. 3, pp. 73–135). Dunod.
- Benzécri, J.-P. (1973a). *L'Analyse des données: la taxinomie* (vol. I). Dunod.
- Benzécri, J.-P. (1973b). *L'Analyse des données: l'analyse des correspondances* (vol. II). Dunod.
- Benzécri, J.-P. (1981a). *Analyse statistique des données linguistiques* (vol. 3, pp. 3–45). Dunod.
- Benzécri, J.-P. (Ed.). (1981b). *Pratique de l'analyse des données: Linguistique et lexicologie* (vol. 3). Dunod.
- 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., & Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media.
- Björkelund, A., Bohnet, B., Hafsdell, L., & Nugues, P. (2010). A high-performance syntactic and semantic dependency parser. In *Coling 2010: Demonstration volume* (pp. 33–36). Coling 2010 Organizing Committee.
- Bohbot, H., Frontini, F., Luxardo, G., Khemakhem, M., & Romary, L. (2018). Presenting the nénufar project: A diachronic digital edition of the petit larousse illustré. In *GLOBALEX 2018–Globalex Workshop at LREC2018* (pp. 1–6)
- Bojanowski, P., Grave, E., Joulin, A., & 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.
- Bourne, S. R. (1982). *The unix system. International computer science series*. Addison-Wesley.
- Brants, T., Popat, A. C., Xu, P., Och, F. J., & 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, Prague* (pp. 858–867)
- Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Wadsworth.
- Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4), 543–565.
- Brin, S., & 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, S. A., Della Pietra, V. J., & Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2), 263–311.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (vol. 33, pp. 1877–1901). Curran Associates.
- Bullinaria, J. A., & Levy, J. P. (2007). Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods*, 39(3), 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.
- Busa, R. (1996). *Thomae Aquinatis Opera omnia cum hypertextibus in CD-ROM*. Editoria Elettronica Editel.
- 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*.
- Candito, M., Crabbé, B., Denis, P., & Guérin, F. (2009). Analyse syntaxique du français: des constituants aux dépendances. In *Actes de la 16ème conférence sur le Traitement Automatique des Langues Naturelles TALN 2009, Senlis*.
- 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.

- Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. (2017). SemEval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)* (pp. 1–14). Association for Computational Linguistics.
- Charniak, E. (1993). *Statistical language learning*. MIT Press.
- Chen, S. F., & Goodman, J. (1998). An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Harvard University, Cambridge.
- Chollet, F. (2021). *Deep learning with Python* (2nd edn.). Manning Publications.
- Chrupała, G. (2006). Simple data-driven context-sensitive lemmatization. In *Proceedings of SEPLN, Zaragoza*.
- Church, K. W., & Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1), 22–29.
- Church, K. W., & Mercer, R. L. (1993). Introduction to the special issue on computational linguistics using large corpora. *Computational Linguistics*, 19(1), 1–24.
- Cooper, D. (1999). Corpora: Kwic concordances with Perl. CORPORA Mailing List Archive, Concordancing thread.
- Crystal, D. (1997). *The Cambridge encyclopedia of language* (2nd edn.). Cambridge University Press.
- d'Arc, S. J. (Ed.). (1970). *Concordance de la Bible, Nouveau testament*. Éditions du Cerf – Desclées De Brouwer.
- de la Briandais, R. (1959). File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference* (pp. 295–298). AFIPS.
- de Marneffe, M.-C., Manning, C. D., Nivre, J., & Zeman, D. (2021). Universal dependencies. *Computational Linguistics*, 47(2), 255–308.
- Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6), 391–407.
- Dermatas, E., & Kokkinakis, G. K. (1995). Automatic stochastic tagging of natural language texts. *Computational Linguistics*, 21(2), 137–163.
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (pp. 4171–4186). Association for Computational Linguistics.
- Dolan, W. B., & Brockett, C. (2005). Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*.
- Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. In *Proceedings of the 4th International Conference on Learning Representations* (pp. 1–4).
- Ducrot, O., & Schaeffer, J.-M. (Eds.). (1995). *Nouveau dictionnaire encyclopédique des sciences du langage*. Éditions du Seuil.
- 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.
- Déjean, H. (1998). Morphemes as necessary concept for structures discovery from untagged corpora. In *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Natural Language Learning* (pp. 295–298). Macquarie University.
- Eckart, C., & Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3), 211–218.
- Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179–211.
- Estoup, J.-B. (1912). *Gammes sténographiques: Recueil de textes choisis pour l'acquisition méthodique de la vitesse* (3e edn.). Institut sténographique.
- Fano, R. M. (1961). *Transmission of information: A statistical theory of communications*. MIT Press.

- 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., & Kucera, H. (1982). *Frequency analysis of english usage*. Houghton Mifflin.
- Franz, A., & Brants, T. (2006). All our n-gram are belong to you. Retrieved November 7, 2013, from <https://googleresearch.blogspot.se/2006/08/all-our-n-gram-are-belong-to-you.html>
- Fredholm, I. (1903). Sur une classe d'équations fonctionnelles. *Acta Mathematica*, 27, 365–390.
- Friedl, J. E. F. (2006). *Mastering regular expressions* (3rd edn.). O'Reilly.
- Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29, 1189–1232.
- Gage, P. (1994). A new algorithm for data compression. *The C User Journal*, 12(2), 23–38.
- Gale, W. A., & Church, K. W. (1993). A program for aligning sentences in bilingual corpora. *Computational Linguistics*, 19(1), 75–102.
- Galton, F. (1886). Regression towards mediocrity in hereditary stature. *Journal of the Anthropological Institute*, 15, 246–263.
- Good, I. J. (1953). The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(16), 237–264.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. The MIT Press.
- Google (2019). WordPieceTokenizer in BERT. Retrieved February 05, 2023, from <https://github.com/google-research/bert/blob/master/tokenization.py#L300-L359>
- Goyvaerts, J., & Levithan, S. (2012). *Regular expressions cookbook* (2nd edn.). O'Reilly Media.
- Grefenstette, G., & Tapanainen, P. (1994). What is a word, what is a sentence? Problems of tokenization. MLTT Technical Report 4, Xerox.
- Grus, J. (2019). *Data science from scratch: First principles with Python* (2nd edn.). O'Reilly Media.
- Guilbaud, A. (2017). L'ENCCRE, édition numérique collaborative et critique de l'Encyclopédie. In *Recherches sur Diderot et sur l'Encyclopédie* (pp. 5–22)
- Guillaume, B., de Marneffe, M.-C., & Perrier, G. (2019). Conversion et améliorations de corpus du français annotés en universal dependencies [conversion and improvement of universal dependencies french corpora]. *Traitement Automatique des Langues*, 60(2), 71–95.
- Guo, P. (2014). Python is now the most popular introductory teaching language at top U.S. universities. Retrieved November 23, 2015, from <https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext>
- Guthrie, R. (2023). Bi-LSTM conditional random field discussion. [https://pytorch.org/tutorials/beginner/nlp/advanced\\_tutorial.html](https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html).
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 10–18.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2nd edn.). Springer.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778). IEEE Computer Society.
- Heafield, K. (2011). KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation* (pp. 187–197). Association for Computational Linguistics.
- Hinton, G. (2012). RMSProp: Divide the gradient by a running average of its recent magnitude. *Coursera: Neural Networks for Machine Learning*, 4, 26–31.
- Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition* (vol. 1, pp. 278–282). IEEE.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
- Hoerl, A. E. (1962). Application of ridge analysis to regression problems. *Chemical Engineering Progress*, 58(3), 54–59.

- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to automata theory, languages, and computation* (3rd edn.). Addison-Wesley.
- Hornby, A. S. (Ed.). (1974). *Oxford advanced learner's dictionary of current english* (3rd edn.). Oxford University Press.
- Huang, J., Gao, J., Miao, J., Li, X., Wang, K., & Behr, F. (2010). Exploring web scale language models for search query processing. In *Proceedings of the 19th International World Wide Web Conference* (pp. 451–460). Raleigh.
- Ide, N., & Véronis, J. (1995). *Text encoding initiative: Background and context*. Kluwer Academic.
- Imbs, P., & Quemada, B. (Eds.), (1971–1994). *Trésor de la langue française. Dictionnaire de la langue française du XIXe et du XXe siècle (1789–1960)* (16 volumes). Éditions du CNRS puis Gallimard.
- ISO/IEC. (2016). *Information technology – Document description and processing languages – Office open XML file formats* (volume ISO/IEC 29500-1:2016(E)). ISO/IEC.
- Iyer, S., Dandekar, N., & Csernai, K. (2017). First Quora dataset release: Question pairs. <https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs>
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An introduction to statistical learning: With applications in R* (2nd edn.). Springer.
- Jelinek, F. (1990). Self-organized language modeling for speech recognition. In A. Waibel & K.-F. Lee (Eds.), *Readings in speech recognition*. Morgan Kaufmann. Reprinted from an IBM Report, 1985.
- Jelinek, F. (1997). *Statistical methods for speech recognition*. MIT Press.
- Jelinek, F., & Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In E. S. Gelsema & L. N. Kanal (Eds.), *Pattern recognition in practice* (pp. 38–397). North-Holland.
- Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., & Sayed, W. E. (2023). Mistral 7b. <https://doi.org/10.48550/arXiv.2310.06825>.
- Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535–547.
- Jordan, C. (1874). Mémoire sur les formes bilinéaires. *Journal de Mathématiques Pures et Appliquées, Deuxième Série*, 19, 35–54.
- Jurafsky, D., & Martin, J. H. (2008). *Speech and language processing, an introduction to natural language processing, computational linguistics, and speech recognition* (2nd edn.). Pearson Education.
- Kaeding, F. W. (1897). *Häufigkeitswörterbuch der deutschen Sprache*. Selbstverlag des Herausgebers.
- Kaplan, R. M., & Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3), 331–378.
- Karpathy, A. (2022). mingpt. <https://github.com/karpathy/minGPT/blob/master/mingpt/bpe.py>
- Karttunen, L. (1983). KIMMO: A general morphological processor. *Texas Linguistic Forum*, 22, 163–186.
- Karttunen, L., Kaplan, R. M., & Zaenen, A. (1992). Two-level morphology with composition. In *Proceedings of the 15th Conference on Computational Linguistics, COLING-92* (vol. 1, pp. 141–148). Nantes.
- 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., & 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), Helsinki* (vol. II, pp. 205–210).
- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv:1412.6980.
- Kiraz, G. A. (2001). *Computational nonlinear morphology: With emphasis on semitic languages. Studies in natural language processing*. Cambridge University Press.

- Kiss, T., & Strunk, J. (2006). Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4), 485–525.
- Klang, M. (2023). Hashing with MD5. Personal communication.
- Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In C. E. Shannon & J. McCarthy (Eds.), *Automata studies* (pp. 3–42). Princeton University Press.
- Knuth, D. E. (1986). *The TeXbook*. Addison-Wesley.
- Kong, Q., Siauw, T., & Bayen, A. (2020). *Python programming and numerical methods: A guide for engineers and scientists*. Academic Press.
- Kornai, A. (Ed.). (1999). *Extended finite state models of language. Studies in natural language processing*. Cambridge University Press.
- Koskenniemi, K. (1983). Two-level morphology: A general computation model for word-form recognition and production. Technical Report 11, University of Helsinki, Department of General Linguistics.
- Kudo, T. (2017). SentencePiece. Retrieved on June 02, 2023, from <https://github.com/google/sentencepiece>
- Kudo, T. (2018). Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 66–75). Association for Computational Linguistics.
- Kudo, T., & Richardson, J. (2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (pp. 66–71). Association for Computational Linguistics.
- Kudoh, T., & Matsumoto, Y. (2000). Use of support vector learning for chunk identification. In *Proceedings of CoNLL-2000 and LLL-2000, Lisbon* (pp. 142–144)
- Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, 22(1), 79–86.
- Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML-01)* (pp. 282–289). Morgan Kaufmann Publishers.
- Lallot, J. (Ed.). (1998). *La grammaire de Denys le Thrace* (2e edn.). CNRS Éditions, Collection Science du langage. Text in Greek, translated in French by Jean Lallot.
- Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., & Dyer, C. (2016). Neural architectures for named entity recognition. In K. Knight, A. Nenkova, & O. Rambow (Eds.), *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 260–270). Association for Computational Linguistics.
- Laplace, P. (1820). *Théorie analytique des probabilités* (3rd edn.). Coursier.
- Le Cun, Y. (1987). *Modèle connexionniste de l'apprentissage*. Ph.D. Thesis, Université Paris 6.
- Legendre, A.-M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes*. Firmin Didot.
- Lewis, D. D., Yang, Y., Rose, T. G., & Li, F. (2004). RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5, 361–397.
- Lin, T., Wang, Y., Liu, X., & Qiu, X. (2022). A survey of transformers. *AI Open*, 3, 111–132.
- Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. <http://arxiv.org/abs/1907.11692>.
- Longpre, S., Hou, L., Vu, T., Webson, A., Chung, H. W., Tay, Y., Zhou, D., Le, Q. V., Zoph, B., Wei, J., et al. (2023). The flan collection: Designing data and methods for effective instruction tuning. arXiv:2301.13688.
- Lutz, M. (2013). *Learning Python* (5th edn.). O'Reilly Media.
- Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association*

- for Computational Linguistics: Human Language Technologies (pp. 142–150). Association for Computational Linguistics.
- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.
- Manning, C. D., & Schütze, H. (1999). *Foundations of statistical natural language processing*. MIT Press.
- Marcus, M., Marcinkiewicz, M. A., & Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2), 313–330.
- Matthes, E. (2019). *Python crash course: A hands-on, project-based introduction to programming* (2nd edn.). No Starch Press.
- Mauldin, M. L., & 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.
- McKinney, W. (2010). Data structures for statistical computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61).
- McKinney, W. (2022). *Python for data analysis: Data wrangling with pandas, NumPy, and Jupyter* (3rd edn.). O'Reilly Media.
- McMahon, J. G., & Smith, F. J. (1996). Improving statistical language models performance with automatically generated word hierarchies. *Computational Linguistics*, 22(2), 217–247.
- Merialdo, B. (1994). Tagging English text with a probabilistic model. *Computational Linguistics*, 20(2), 155–171.
- Mikheev, A. (2002). Periods, capitalized words, etc. *Computational Linguistics*, 28(3), 289–318.
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013a). Efficient estimation of word representations in vector space. In *First International Conference on Learning Representations (ICLR2013)*, *Workshop Proceedings*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems* (vol. 26, pp. 3111–3119). Curran Associates.
- Mohri, M., Pereira, F. C. N., & Riley, M. (1998). A rational design for a weighted finite-state transducer library. In D. Wood, & S. Yu (Eds.), *Automata Implementation. Second International Workshop on Implementing Automata, WIA '97, London, Ontario, September 1997. Revised Papers. Lecture notes in computer science* (vol. 1436, pp. 144–158). Springer.
- Mohri, M., Pereira, F. C. N., & Riley, M. (2000). The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1), 17–32.
- Monachini, M., & Calzolari, N. (1996). Synopsis and comparison of morphosyntactic phenomena encoded in lexicons and corpora: A common proposal and applications to European languages. Technical Report, Istituto di Linguistica Computazionale del CNR, Pisa. EAGLES Document EAG-CLWG-MORPHSYN/R.
- Moore, E. H. (1920). On the reciprocal of the general algebraic matrix, abstract. *Bulletin of the American Mathematical Society*, 26, 394–395.
- Murphy, K. P. (2022). *Probabilistic machine learning: An introduction*. MIT Press.
- Ney, H., Essén, U., & Kneser, R. (1994). On structuring probabilistic dependences in stochastic language modelling. *Computer Speech and Language*, 8(1), 1–38.
- Nivre, J., Agić, Ž., Ahrenberg, L., Antonsen, L., Aranzabe, M. J., Asahara, M., Ateyah, L., Attia, M., Atutxa, A., Augustinus, L., Badmaeva, E., Ballesteros, M., Banerjee, E., Bank, S., Barbu Mititelu, V., Bauer, J., Bengoetxea, K., Bhat, R. A., Bick, et al. (2017). Universal dependencies 2.1. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.
- Norvig, P. (2007). How to write a spelling corrector. Retrieved November 30, 2015, from <https://norvig.com/spell-correct.html>
- Norvig, P. (2009). Natural language corpus data. In *Beautiful data: The stories behind elegant data solutions* (pp. 219–242). O'Reilly Media.
- Oliphant, T. E. (2015). *Guide to NumPy* (2nd edn.). CreateSpace Independent Publishing Platform.

- Orwell, G. (1949). *Nineteen eighty-four*. Secker and Warburg.
- 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.
- Palomar, M., Civit, M., Díaz, A., Moreno, L., Bisbal, E., Aranzabe, M., Ageno, A., Martí, M. A., & Navarro, B. (2004). 3LB: Construcción de una base de datos de árboles sintáctico-semánticos para el catalán, euskera y español. In *XX Congreso de la Sociedad Española para el Procesamiento del Lenguaje Natural (SEPLN), Barcelona* (pp. 81–88).
- Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: A method for automatic evaluation of machine translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics, Philadelphia* (pp. 311–318).
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems* (vol. 32, pp. 8024–8035). Curran Associates.
- 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., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1532–1543). Association for Computational Linguistics.
- Pérennou, G., & de Calmès, M. (1987). BDLex lexical data and knowledge base of spoken and written French. In *European Conference on Speech Technology* (pp. 393–396).
- Petrov, S., Das, D., & McDonald, R. (2012). A universal part-of-speech tagset. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012), Istanbul* (pp. 2089–2096).
- Petruszewycz, M. (1973). L'histoire de la loi d'Estoup-Zipf: documents. *Mathématiques et Sciences Humaines*, 44, 41–56.
- Procter, P. (Ed.). (1978). *Longman dictionary of contemporary english*. Longman.
- Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1), 145–151.
- Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1), 81–106.
- Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. Morgan Kaufmann.
- Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*.
- Rajpurkar, P., Jia, R., & Liang, P. (2018). Know what you don't know: Unanswerable questions for SQuAD. In I. Gurevych & Y. Miyao (Eds.), *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (pp. 784–789). Association for Computational Linguistics.
- Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ questions for machine comprehension of text. In J. Su, K. Duh, & X. Carreras (Eds.), *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (pp. 2383–2392). Association for Computational Linguistics.
- Ramshaw, L. A., & Marcus, M. P. (1995). Text chunking using transformation-based learning. In D. Yarowsky & K. Church (Eds.), *Proceedings of the Third Workshop on Very Large Corpora, Cambridge* (pp. 82–94).
- Raschka, S., Liu, H. Y., & Mirjalili, V. (2022). *Machine learning with PyTorch and Scikit-learn*. Packt Publishing.

- Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. In E. Brill & K. Church (Eds.), *Proceedings of the Conference on Empirical Methods in Natural Language Processing, Philadelphia* (pp. 133–142).
- Ray, E. T. (2003). *Learning XML* (2nd edn.). O'Reilly.
- Reimers, N., & Gurevych, I. (2019). Sentence-BERT: sentence embeddings using siamese BERT-networks. In K. Inui, J. Jiang, V. ng, & X. Wan (Eds.), *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* (pp. 3982–3992). Association for Computational Linguistics, Hong Kong.
- Ren, X., Zhou, P., Meng, X., Huang, X., Wang, Y., Wang, W., Li, P., Zhang, X., Podolskiy, A., Arshinov, G., Bout, A., Piontovskaya, I., Wei, J., Jiang, X., Su, T., Liu, Q., & Yao, J. (2023). Pangu- $\Sigma$ : Towards trillion parameter language model with sparse heterogeneous computing. <https://doi.org/10.48550/arXiv.2303.10845>.
- Reynar, J. C. (1998). *Topic Segmentation: Algorithms and Applications*. Ph.D. Thesis, University of Pennsylvania, Philadelphia.
- Reynar, J. C., & Ratnaparkhi, A. (1997). A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the Fifth Conference on Applied Natural Language Processing, Washington* (pp. 16–19).
- Ritchie, G. D., Russell, G. J., Black, A. W., & Pulman, S. G. (1992). *Computational morphology. Practical mechanisms for the english lexicon*. MIT Press.
- Roche, E., & Schabes, Y. (1995). Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2), 227–253.
- Roche, E., & Schabes, Y. (Eds.). (1997). *Finite-state language processing*. MIT Press.
- 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., & Rouse, M. A. (1974). The verbal concordance to the scriptures. *Archivum Fratrum Praedicatorum*, 44, 5–30.
- Ruder, S. (2017). An overview of gradient descent optimization algorithms. <http://arxiv.org/abs/1609.04747>.
- Rush, A. (2018). The annotated transformer. In E. L. Park, M. Hagiwara, D. Milajevs, & L. Tan (Eds.), *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)* (pp. 52–60). Association for Computational Linguistics.
- Salton, G. (1988). *Automatic text processing: The transformation, analysis, and retrieval of information by computer*. Addison-Wesley.
- Salton, G., & Buckley, C. (1987). Term weighting approaches in automatic text retrieval. Technical Report TR87-881, Department of Computer Science, Cornell University, Ithaca.
- Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. In *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing @ NeurIPS 2019*.
- Saporta, G. (2011). *Probabilités, analyse des données et statistiques* (3rd edn.). Éditions Technip.
- Schuster, M., & Nakajima, K. (2012). Japanese and Korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto* (pp. 5149–5152).
- Sennrich, R., Haddow, B., & Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1715–1725). Association for Computational Linguistics.
- Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27, 398–403, 623–656.
- Shao, Y., Hardmeier, C., & Nivre, J. (2018). Universal word segmentation: Implementation and interpretation. *Transactions of the Association for Computational Linguistics*, 6, 421–435.
- Shazeer, N. (2020). Glu variants improve transformer. <https://arxiv.org/abs/2002.05202>.

- Silveira, N., Dozat, T., de Marneffe, M.-C., Bowman, S., Connor, M., Bauer, J., & Manning, C. D. (2014). A gold standard dependency corpus for English. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*.
- Simone, R. (2007). *Fondamenti di linguistica* (10th edn.). Laterza.
- Sinclair, J. (Ed.). (1987). *Collins COBUILD english language dictionary*. Collins.
- Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., & Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (pp. 1631–1642). Association for Computational Linguistics.
- Spärck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1), 11–21.
- Sproat, R. (1992). *Morphology and computation*. MIT Press.
- Stevens, E., Antiga, L., & Viehmann, T. (2020). *Deep learning with PyTorch*. Manning Publications.
- Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. In A. Korhonen, D. Traum, & L. Màrquez (Eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (pp. 3645–3650). Association for Computational Linguistics.
- Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2024). Roformer: Enhanced transformer with rotary position embedding. <https://doi.org/10.1016/j.neucom.2023.127063>.
- Suits, D. B. (1957). Use of dummy variables in regression equations. *Journal of the American Statistical Association*, 52(280), 548–551.
- Sutton, C., & McCallum, A. (2011). An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4), 267–373.
- Taylor, W. L. (1953). “cloze procedure”: A new tool for measuring readability. *Journalism Quarterly*, 30, 415–433.
- TEI Consortium, e. (2023). TEI P5: Guidelines for electronic text encoding and interchange. Retrieved September 29, 2023, from <https://www.tei-c.org/Guidelines/P5/>.
- The Unicode Consortium. (2012). *The unicode standard, version 6.1 – Core specification*. Unicode Consortium, Mountain View.
- The Unicode Consortium. (2022). *The unicode standard, version 15.0 – Core specification*. Unicode Consortium, Mountain View.
- Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM*, 11(6), 419–422.
- Tjong Kim Sang, E. F. (2002). Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *Proceedings of CoNLL-2002, Taipei* (pp. 155–158).
- Tjong Kim Sang, E. F., & Buchholz, S. (2000). Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL-2000 and LLL-2000, Lisbon* (pp. 127–132).
- Tjong Kim Sang, E. F., & De Meulder, F. (2003). Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of CoNLL-2003, Edmonton* (pp. 142–147).
- Tjong Kim Sang, E. F., & Veenstra, J. (1999). Representing text chunks. In *Ninth Conference of the European Chapter of the ACL, Bergen* (pp. 173–179).
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023a). Llama: Open and efficient foundation language models. arxiv:2302.13971.
- Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, et al. (2023b). Llama 2: Open foundation and fine-tuned chat models. <https://doi.org/10.48550/arXiv.2307.09288>.
- Tunstall, L., Von Werra, L., & Wolf, T. (2022). *Natural language processing with transformers, Revised Edition*. O'Reilly Media.
- van Noord, G., & Gerdemann, D. (2001). An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt & H. Jürgensen (Eds.), *Automata*

- Implementation. 4th International Workshop on Implementing Automata, WIA'99, Potsdam, Germany, July 1999, Revised Papers. Lecture notes in computer science* (vol. 2214, pp. 122–139). Springer.
- van Rossum, G., Warsaw, B., & Coghlan, N. (2013). PEP 0008 – Style guide for Python code. Retrieved November 23, 2015, from <https://www.python.org/dev/peps/pep-0008/>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems* (vol. 30). Curran Associates.
- Vergne, J. (1999). *Étude et modélisation de la syntaxe des langues à l'aide de l'ordinateur. Analyse syntaxique automatique non combinatoire. Synthèse et résultats*. Habilitation à diriger des recherches, Université de Caen.
- 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.
- Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2), 260–267.
- Voutilainen, A., & Järvinen, T. (1995). Specifying a shallow grammatical representation for parsing purposes. In *Proceedings of the Seventh Conference of the European Chapter of the ACL, Dublin* (pp. 210–214).
- Wall, L., Christiansen, T., & Orwant, J. (2000). *Programming Perl* (3rd edn.). O'Reilly.
- Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2018). GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP* (pp. 353–355). Association for Computational Linguistics.
- Wang, K., Thrasher, C., Viegas, E., Li, X., & Hsu, B. (2010). An overview of Microsoft web n-gram corpus and applications. In *Proceedings of the NAACL HLT 2010: Demonstration Session, Los Angeles* (pp. 45–48).
- Wang, W., Zhang, L. S., Zinsmaier, A. K., Patterson, G., Leptich, E. J., & Tatiana A. Yatskivych, S. L. S., Gibboni, R., Pace, E., Luo, H., Zhang, J., Yang, S., & Bao, S. (2019). Neuroinflammation mediates noise-induced synaptic imbalance and tinnitus in rodent models. *PLoS Biology*, 17, e3000307.
- Warstadt, A., Singh, A., & Bowman, S. R. (2018). Neural network acceptability judgments. arXiv:1805.12471.
- Wenzek, G., Lachaux, M.-A., Conneau, A., Chaudhary, V., Guzmán, F., Joulin, A., & Grave, E. (2020). CCNet: Extracting high quality monolingual datasets from web crawl data. In N. Calzolari, F. Béchet, P. Blache, K. Choukri, C. Cieri, T. Declerck, S. Goggi, H. Isahara, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, & S. Piperidis (Eds.), *Proceedings of the Twelfth Language Resources and Evaluation Conference* (pp. 4003–4012). European Language Resources Association.
- Whistler, K., & Scherer, M. (2023). Unicode collation algorithm. Unicode Technical Standard 10, The Unicode Consortium.
- Williams, A., Nangia, N., & Bowman, S. (2018). A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (pp. 1112–1122). Association for Computational Linguistics.
- Witten, I. H., & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques* (2nd edn.). Morgan Kaufmann.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., & Rush, A. (2020). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical*

- Methods in Natural Language Processing: System Demonstrations* (pp. 38–45). Association for Computational Linguistics.
- Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. CoRR, abs/1609.08144.
- Yu, H.-F., Ho, C.-H., Juan, Y.-C., & Lin, C.-J. (2013). LibShortText: A library for short-text classification and analysis. Retrieved November 1, 2013, from <https://www.csie.ntu.edu.tw/~cjlin/libshorttext>
- 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., & Robertson, S. (2004). Microsoft Cambridge at TREC-13: Web and HARD tracks. In *Proceedings of TREC-2004, Gaithersburg*.
- Zhang, B., & Sennrich, R. (2019). Root mean square layer normalization. In *Advances in Neural Information Processing Systems, Vancouver* (vol. 32).

# Index

## A

Abadi, M., 208  
Abeillé, A., 334, 335  
Activation function, 199  
Adam, 186  
Adjective, 326  
Adverb, 326  
Aelius Donatus, 325  
Alignment, 474  
Allauzen, C., 83  
Allomorph, 342  
Ambiguity, 4, 345  
American Standard Code for Information Interchange (ASCII), 85  
Annotated corpus, 48  
Annotation schemes, 85  
Antworth, E.L., 346, 347  
Approximate string matching, 77  
*Ars grammatica*, 325  
Attention, 426  
Attribute, 150  
Attribute domain, 150  
Auxiliary verb, 326

## B

Back-off, 267  
Backpropagation, 201  
Baeza-Yates, R., 244, 251  
Bag-of-word model, 246  
Bahdanau, D., 480  
Ba, J., 186  
Ba, J.L., 435  
Bank of English, 277, 278  
Baseline, 382, 383

Basic multilingual plane, 88  
Batch gradient descent, 170  
Baudot, É., 86  
BDLex, 338  
Beesley, K.R., 348, 353  
Beltrami, E., 287  
Bentivogli, L., 461  
Bentley, J., 242  
Benzécri, F., 294  
Benzécri, J.-P., 294, 322  
Berkson, J., 180, 192  
Bible studies, 50  
Bidirectional encoder representations from transformers (BERT), 449  
Bigram, 254  
Bilingual evaluation understudy (BLEU), 482  
Binary cross-entropy, 183  
BIOES scheme, 414  
BIO scheme, 413  
Bird, S., 8  
Björkelund, A., 353  
Bohbot, H., 339  
Bojanowski, P., 322  
Boltzmann distribution, 221  
Boscovich, R.J., 165  
Bourne, S.R., 242  
Boustrophedon, 229  
Branching factor, 272  
Brants, T., 267, 270  
Breiman, L., 158  
Brill tagger, 415  
Brill, E., 422, 423  
Brin, S., 248  
Broadcasting, 127  
Brockett, C., 461

- Brown, P.F., 473, 476  
 Brown, T., 498  
 Buchholz, S., 414, 415  
 Buckley, C., 248  
 Bullinaria, J.A., 299  
 Busa, R., 82, 83  
 Byte order mark, 94  
 Byte-pair encoding (BPE), 357, 379
- C**  
 Calmès, M. de, 338  
 Calzolari, N., 329, 330  
 Canadian Hansard, 473  
 Candito, M., 334  
 Car, D., 461  
 CART, 158  
 Categorical cross entropy, 220  
 Cauchy, A., 168  
 CBOW, 304  
 Chain rule, 256  
 Character class, 63  
 Character range, 63  
 Charniak, E., 382  
 Chen, S.F., 283  
 Chollet, F., 208, 275  
 Chrupala, G., 353  
 Chunk annotation schemes, 414  
 Chunking, 409  
 Church, K.W., 50, 51, 253, 276, 475  
 Clément, L., 334, 335  
 Closed class, 326  
 Closed vocabulary, 259  
 Closure operator, 60  
 Cloze test, 305, 450  
 Code, 141  
 Collocation, 276  
 Common Crawl, 82  
 Composition, 342  
 Compounding, 342  
 Concatenative morphology, 341  
 Concordance, 49, 74, 82  
 Conditional random fields, 417  
 Confusion matrix, 385  
 Conjunction, 326  
 CoNLL format, 333  
 CoNLL reader, 334  
 Cooper, D., 75  
 Copula, 328  
 Corpora, 47  
 Corpus, 47  
 Corpus balancing, 48  
 Correspondence analysis, 294  
 Crampon, A., 50
- Cross-attention, 480  
 Cross-entropy, 148, 271  
 Cross-perplexity, 149  
 Cross-validation, 158, 258  
 Crystal, D., 48
- D**  
 Decision tree, 149  
 Decoder, 425, 473, 495  
 Decorators, 38  
 Deerwester, S., 293, 294  
 Déjean, H., 355  
 de la Briandais, R., 339  
 de Marneffe, M.-C., 49, 334  
 De Meulder, F., 415, 416  
 Derivation, 342  
 Dermatas, E., 329  
 Determiner, 326  
 Deterministic automata, 54  
 Development set, 158, 258  
 Devlin, J., 446, 449, 451, 453, 455, 460, 461, 463, 464, 469  
 Dictionary, 339  
 Dimensionality reduction, 286  
 Dionysius Thrax, 325, 353  
 Discount, 263  
 Ditransitive, 328  
 DocBook, 104  
 Docstrings, 41  
 Document categorization, 249  
 Document indexing, 243  
 Document ranking, 248  
 Document retrieval, 243  
 Document type definition (DTD), 103  
 Dolan, W.B., 461  
 Dozat, T., 186  
 Dropout, 405  
 Ducrot, O., 353  
 Dumais, S.T., 293  
 Dunning, T., 278, 283
- E**  
 Eckart, C., 287  
 Edit operation, 77  
 Elman, J.L., 398  
 Embeddings, 395  
 Encoder, 425  
 Encoder-decoder, 473, 476  
 Encoding, 85  
 English Web Treebank (EWT), 334, 391  
 Entropy, 141, 142  
 Entropy rate, 271

- Epoch, 170  
Escape character, 61  
Estoup, J.-B., 83  
European Language Resources Association (ELRA), 48  
European Parliament proceedings, 473  
Evaluation of classification systems, 193  
Expectation-maximization, 371  
Extensible hypertext markup language (XHTML), 104  
Extensible Markup Language (XML), 86
- F**  
Faiss, 252  
Fallout, 195  
Fano, R., 299  
Fano, R.M., 276  
Feature, 326  
Feature vector, 176  
Feed-forward neural network, 197  
Ferrucci, D.A., 3  
Finite-state automata, 52  
Finite-state transducer, 349  
Flaubert, G., 142, 143, 149  
*F*-measure, 195  
Frank, E., 160  
FranText, 82  
Franz, A., 270  
Fredholm, I., 172  
Friedl, J.E.F., 83  
Friedman, J.H., 159  
FST composition, 352  
FST inversion, 352  
FST operations, 352  
Functional programming, 43
- G**  
Gage, P., 355, 357, 363  
Gale, W.A., 475  
Galton, F., 196  
General Language Understanding Evaluation (GLUE), 460  
Generators, 32  
Genesis, 48  
Gerdemann, D., 83, 353  
GloVe, 300, 395, 427  
Gold standard, 49  
Goodfellow, I., 196, 228  
Good, I.J., 263  
Goodman, J., 283  
Good–Turing estimation, 263, 264  
Google, 248
- Google Translate, 3  
Gospel of John, 50  
Goyvaerts, J., 83  
GPT-2, 362, 363  
Gradient ascent, 182  
Gradient boosting, 159  
Gradient descent, 167  
Grammar checker, 2  
Grammatical feature, 326, 327  
Grammatical morpheme, 340  
Grefenstette, G., 237, 238  
Grep, 83  
Group annotation, 411  
Group annotation schemes, 414  
Grus, J., 196  
Guilbaud, A., 339  
Guillaume, B., 385  
Guo, P., 9  
Gurevych, I., 465  
Guthrie, R., 421
- H**  
Hadamard product, 127  
Hall, M., 160  
Hanks, P., 276  
Harmonic mean, 195  
Hashing trick, 317  
Hastie, T., 196  
Hazel, P., 83  
Heafield, K., 283  
Heaviside function, 178  
He, K., 435  
Hinton, G., 186  
Hochreiter, S., 406  
Hoerl, A.E., 172  
Hopcroft, J.E., 54, 56, 59, 83  
Hornby, A.S., 337  
Ho, T.K., 159  
Huang, J., 270  
Huffman code, 145, 148  
Huffman coding, 144  
Huffman tree, 144  
Hugging Face, 377, 447  
Hugh of St-Cher, 50  
Hugo, V., 148  
Hyperplane, 176
- I**  
IBM Watson, 3  
ID3, 152, 159  
Ide, N., 329, 330  
Imbs, P., 82

IMDB corpus, 249  
 Indexing, 251  
 Induction of decision trees, 152  
 Inflection, 342  
 Information retrieval, 2, 243, 251  
 Information theory, 141  
 Interactive voice response, 3  
 Inter-annotator agreement, 49  
 International components for Unicode (ICU),  
     99  
 Internet crawler, 243  
 Intransitive, 328  
 Inverse document frequency, 248  
 Inverted index, 244  
 IOB scheme, 411, 414  
 IPython, 45  
 ISO-8859-1, 87  
 ISO-8859-15, 87  
 Iterators, 32  
 Iyer, S., 100, 461

**J**

Järvinen, T., 381  
 Jacobian, 204  
 James, G., 196, 228  
 Jelinek, F., 254, 266  
 Jeopardy, 3  
 Jiang, A.Q., 497  
 Johnson, J., 252  
 Jordan, C., 287  
 Jurafsky, D., 283

**K**

Kaeding, F.W., 83  
 Kaplan, R.M., 352  
 Karpathy, A., 379  
 Karttunen, L., 347, 348, 353  
 Katz's back-off, 268  
 Katz, S.M., 264, 267, 268  
 Kay, M., 352  
 KenLM, 283  
 Keras, 208, 228  
 Kernighan, M.D., 78  
 King James version, 50  
 Kingma, D.P., 186  
 Kiraz, G.A., 353  
 Kiss, T., 238  
 Klang, M., 318  
 Kleene, S.C., 83  
 Kleene star, 60  
 Kneser-Ney smoothing, 269  
 Knuth, D.E., 103

Kokkinakis, G.K., 329  
 Kong, Q., 46  
 Kornai, A., 353  
 Koskenniemi, K., 347  
 Kubrick, S., 1  
 Kudo, T., 355, 369, 376, 379, 423  
 Kullback-Leibler divergence, 148  
 Kullback, S., 148

**L**

Lafferty, J., 417  
 Lallot, J., 353  
 Lambda expressions, 43  
 Lample, G., 420, 421  
 Language detection, 313  
 Language model, 253  
 Lapis niger, 229  
 Laplace's rule, 261  
 Laplace, P.-S., 261  
 Latent semantic indexing, 294  
 Latin 1 character set, 87  
 Latin 9 character set, 87  
 Lazy matching, 62  
 Learning rate, 169  
 Least absolute deviation (LAD), 165  
 Least squares, 164  
 Leave-one-out cross-validation, 159  
 Leavitt, J.R.R., 248  
 Le Cun, Y., 203  
 Legendre, A.-M., 164  
 Leibler, R.A., 148  
 Lemma, 344  
 Lemmatization, 344

Letter tree, 339

Levithan, S., 83  
 Levy, J.P., 299  
 Lewis, D.D., 249  
 Lexical morpheme, 340  
 Lexicon, 337  
 Likelihood ratio, 278  
 Linear classification, 173  
 Linear interpolation, 266  
 Linear regression, 161, 163  
 Linear separability, 176  
 Linguistic Data Consortium (LDC), 48, 82  
 Link verb, 328  
 Lin, T., 447  
 Llama, 497  
 Locale, 95  
 Logistic curve, 180  
 Logistic loss, 183  
 Logistic regression, 161, 178, 180, 235, 388  
 Logit, 181, 215

- Log-likelihood, 182  
Log likelihood ratio, 281  
Log-loss, 183  
Log-sum-exp trick, 215  
Longest match, 61  
Long short-term memory (LSTM), 406  
Lonpre, S., 498  
Lookahead, 76  
Lookbehind, 76  
Loss function, 164, 165  
Luther's Bible, 50  
Lutz, M., 46
- M**  
Maas, A.L., 249, 465  
Machine-learning, 141, 161, 197  
Machine-learning library, 156  
Machine-learning techniques, 150  
Machine translation, 3  
Macro average, 195  
Manning, C.D., 159, 244, 251, 283  
Marcus, M., 236, 238, 411, 423  
Markup language, 48, 103  
Martin, J.H., 283  
Matsumoto, Y., 423  
Matthes, E., 46  
Mauldin, M.L., 248  
Maximum likelihood estimation (MLE), 256  
McCallum, A., 417, 418, 423  
McKinney, W., 102, 109  
McMahon, J.G., 4  
Memo function, 38  
Memoization, 38  
Mercer, R.L., 50, 51, 253, 266, 277  
Merialdo, B., 382  
Metacharacters in a character class, 63  
Mikheev, A., 238  
Mikolov, T., 304, 306, 310–313, 322  
Minibatch, 170  
Minimum edit distance, 80  
Mistral, 497  
Models, 5  
Modal verb, 326  
Mohri, M., 83, 353  
Momentum, 186  
Monachini, M., 329, 330  
Mood, 328  
Moore, E.H., 172  
Morfin, M., 294  
Morph, 342  
Morpheme, 340  
Morphology, 325  
Morse code, 144
- Multinomial classification, 178  
Multinomial logistic regression, 192  
Murphy, K.P., 196, 228  
Mutual information, 276, 280, 299
- N**  
NAdam, 186  
Nakajima, K., 355, 364  
Named entity recognition, 409  
Natural Language Toolkit (NLTK), 8, 251, 283  
Negated character class, 63  
Negative log-likelihood (NLL), 183, 258, 370  
Neural networks, 178, 197  
Ney, H., 269  
*n*-gram, 254  
Nineteen Eighty-Four, 148, 149, 239, 243, 256, 258, 260, 264  
Nivre, J., 236  
Nondeterministic automata, 54  
Nonprintable position, 65  
Nonprintable symbol, 65  
Normalization form compatibility decomposition (NFKD), 91  
Norvig, P., 79, 283  
Not linearly separable, 176  
Notre Dame de Paris, 148  
Noun, 326  
Noun chunk, 409  
Noun group, 409  
Null hypothesis, 277  
NumPy, 111
- O**  
Office Open XML, 103  
Oliphant, T.E., 139  
One-hot encoding, 155  
Online learning, 170  
Open class, 326  
OpenNLP, 235  
Open vocabulary, 259  
Opinion mining, 249  
Optimizer, 185  
Order of a tensor, 122  
Orwell, G., 148, 239, 243, 256  
Out-of-vocabulary (OOV) word, 259  
Overgeneration, 351
- P**  
Pérennou, G., 338  
Page, L., 248  
PageRank, 248

Palmer, H.E., 276  
 Palomar, M., 334  
 Pandas, 102  
 Papineni, K., 482  
 Parallel corpora, 473  
 Partes orationis, 325  
 Parts of speech, 325  
 Paszke, A., 208  
 Pedregosa, F., 156, 160, 228, 388  
 Pennington, J., 300, 304, 322, 395, 427  
 Penn Treebank, 236, 238  
 Perceptron, 178  
 Pereira, F., 417  
 Perl compatible regular expressions (PCRE),  
     60, 83  
 Perplexity, 149, 272  
 Petrov, S., 329, 330  
 Petruszewycz, M., 83  
 Positive mutual information, 299  
 POS tagging, 381  
 POS tagging using classifiers, 385  
 POS tagset, 329  
 Postings list, 243  
 Powerset, 373  
 Precision, 194  
 Predefined character classes, 64  
 Preposition, 326  
 Principal component analysis (PCA), 292  
 Procter, P., 337  
 Pronoun, 326  
 Pseudoinverse matrix, 172  
 PyCharm, 45  
 Python, 9  
 Python backreferences, 70  
 Python dictionaries, 23  
 Python lists, 17  
 Python match objects, 73  
 Python pattern matching, 67  
 Python pattern substitution, 70  
 Python sets, 22  
 Python strings, 11  
 Python tuples, 21  
 PyTorch, 111, 208

**Q**

Qian, N., 186  
 Quality of a language model, 270  
 Quemada, B., 82  
 Question answering, 3  
 Quinlan, J.R., 150–152, 154  
 Quora question pairs (QQP), 100

**R**

R, 160, 228  
 Radford, A., 362, 495  
 Rajpurkar, P., 463  
 Ramshaw, L.A., 411, 423  
 Random forests, 159  
 Raschka, S., 228  
 Ratnaparkhi, A., 238, 239, 422  
 Raw strings, 71  
 Ray, E.T., 110  
 Recall, 194  
 Rectified linear unit (ReLU), 199  
 Recurrent neural networks (RNN), 397  
 Regex, 59  
 Regular expression, 59  
 Regularization, 171  
 Reimers, N., 465  
 Ren, X., 472  
 Repetition metacharacter, 60  
 Retriever-reader model, 465  
 Reuters corpus, 249  
 Reynar, J.C., 235, 238, 239  
 Ribeiro-Neto, B., 244, 251  
 Richardson, J., 355  
 Ritchie, G.D., 353  
 RMSprop, 186  
 Roche, E., 59, 83, 353, 422  
 Rosenblatt, F., 178  
 Rotary positional embeddings, 497  
 Rouse, M.A., 83  
 Rouse, R.H., 83  
 Ruder, S., 196, 228  
 Rush, A., 447, 489, 494

**S**

Salammbo, 142, 143, 148, 149  
 Salton, G., 246–248  
 Sanh, V., 466  
 Saporta, G., 196  
 Schütze, H., 159, 283  
 Schabes, Y., 59, 83, 353, 422  
 Schaeffer, J.-M., 353  
 Scherer, M., 97, 98, 109  
 Schmidhuber, J., 406  
 Schuster, M., 355, 364  
 Scikit-learn, 8, 156, 160, 228  
 Scraping, 106  
 Searching edits, 82  
 Segmentation, 229  
 Self-attention, 425, 426  
 Sennrich, R., 355, 357, 497

- Sentence detection, 236  
SentencePiece, 377, 379  
Sentence probability, 260  
Sentence segmentation, 236  
Sentiment analysis, 249  
Sequence annotation, 381  
Shannon, C.E., 141  
Shao, Y., 251, 421  
Shazeer, N., 497  
Sigmoid, 180  
Silveira, N., 334, 391  
Simone, R., 341  
Sinclair, J., 82  
Singular value decomposition, 286  
Skipgram, 304  
Slices, 14  
Smith, F.J., 4  
Smoothing, 261  
Socher, R., 461  
Softmax function, 221  
Source language, 474  
Spärck-Jones, K., 248  
SpaCy, 8, 251  
Spam detection, 249  
Sparse data, 261  
Speech transcription, 2  
Spelling checker, 2  
Sproat, R., 353  
SQuAD, 3, 463  
Step size, 169  
Stevens, E., 139, 228  
Stochastic gradient descent, 170  
Strubell, E., 472  
Strunk, J., 238  
Suits, D.B., 155  
Su, J., 497  
Sum of the squared errors, 164  
Supervised classification, 150  
Supervised machine-learning, 150  
Sutton, C., 418, 423  
SwiGLU, 497  
Swiss federal law, 474  
SYSTRAN, 3  
Sœur Jeanne d'Arc, 50
- T**  
Tapanainen, P., 237, 238  
Target language, 474  
Taylor, W.L., 305, 450  
Templatic morphology, 341  
Tense, 328  
Tensor, 122  
Tensorflow, 208
- Test set, 158, 176, 258  
TeX, 103  
Text categorization, 249  
Text Encoding Initiative (TEI), 104, 338  
Text generation, 272  
Text segmentation, 229  
*tf-idf*, 246–248  
Thompson, K., 83  
Tjong Kim Sang, E.F., 413–416  
Token, 229  
Tokenization, 229, 231  
Touvron, H., 496, 497  
Training set, 151, 158, 176, 258  
Transformer, 379, 425  
Transformer decoder, 495  
Transitive, 328  
Trie, 339  
Trigram, 254  
*t*-scores, 277, 281  
Tunstall, L., 472  
Two-level model, 347  
Type, 239
- U**  
Unicode, 87, 109  
Unicode character database, 91  
Unicode character properties, 91  
Unicode collation algorithm, 97  
Unicode transformation format, 94  
Unigram, 254  
Unigram tokenizer, 369  
Universal character set, 88  
Universal Dependencies, 49, 334  
Universal Part-of-Speech Tagset (UPOS), 329  
UTF-16, 94  
UTF-8, 94
- V**  
Validation set, 158, 258  
Van Noord, G., 83, 353  
Van Rossum, G., 10  
Vaswani, A., 379, 425, 429, 431, 438, 446, 473, 477–479, 481, 482, 485, 494  
Vector space model, 246  
Veenstra, J., 413  
Verb, 326  
Verb chunk, 409  
Verb group, 409  
Vergne, J., 382  
Verhulst, P.-F., 180  
Véronis, J., 329, 330  
Viterbi algorithm, 374

Viterbi, A.J., 374

Voice, 328

Voice control, 2

Voutilainen, A., 381

## W

Wang, A., 3, 460, 461

Wang, K., 270

Wang, W., viii

Warstadt, A., 461

Web scraping, 106

Weight vector, 176

Weka, 160

Wenzek, G., 498

Whistler, K., 97, 98, 109

Wikipedia, 8

Williams, A., 461

Witten, I. H., 160

Wolf, T., 447, 465

Word embeddings, 286, 294

Word preference measurements, 276

word2vec, 304, 322

Word type, 239

WordPiece, 364, 451

Wu, Y., 355, 364, 379

## X

XML attribute, 104

XML element, 104

XML entity, 105

XML-TEI, 338

## Y

Young, G., 287

Yu, H.F., 249

## Z

Zampolli, A., 83

Zaragoza, H., 248

Zhang, B., 497

Zip, 33