Natural Language Processing

Jacob Eisenstein

November 13, 2018

Contents

Preface

The goal of this text is focus on a core subset of the natural language processing, unified by the concepts of learning and search. A remarkable number of problems in natural language processing can be solved by a compact set of methods:

Search. Viterbi, CKY, minimum spanning tree, shift-reduce, integer linear programming, beam search.

Learning. Maximum-likelihood estimation, logistic regression, perceptron, expectation-maximization, matrix factorization, backpropagation.

This text explains how these methods work, and how they can be applied to a wide range of tasks: document classification, word sense disambiguation, part-of-speech tagging, named entity recognition, parsing, coreference resolution, relation extraction, discourse analysis, language modeling, and machine translation.

Background

Because natural language processing draws on many different intellectual traditions, almost everyone who approaches it feels underprepared in one way or another. Here is a summary of what is expected, and where you can learn more:

Mathematics and machine learning. The text assumes a background in multivariate calculus and linear algebra: vectors, matrices, derivatives, and partial derivatives. You should also be familiar with probability and statistics. A review of basic probability is found in Appendix A, and a minimal review of numerical optimization is found in Appendix B. For linear algebra, the online course and textbook from Strang (2016) provide an excellent review. Deisenroth et al. (2018) are currently preparing a textbook on *Mathematics for Machine Learning*, a draft can be found online.[1] For an introduction to probabilistic modeling and estimation, see James et al. (2013); for

[1] `https://mml-book.github.io/`

a more advanced and comprehensive discussion of the same material, the classic reference is Hastie et al. (2009).

Linguistics. This book assumes no formal training in linguistics, aside from elementary concepts likes nouns and verbs, which you have probably encountered in the study of English grammar. Ideas from linguistics are introduced throughout the text as needed, including discussions of morphology and syntax (chapter 9), semantics (chapters 12 and 13), and discourse (chapter 16). Linguistic issues also arise in the application-focused chapters 4, 8, and 18. A short guide to linguistics for students of natural language processing is offered by Bender (2013); you are encouraged to start there, and then pick up a more comprehensive introductory textbook (e.g., Akmajian et al., 2010; Fromkin et al., 2013).

Computer science. The book is targeted at computer scientists, who are assumed to have taken introductory courses on the analysis of algorithms and complexity theory. In particular, you should be familiar with asymptotic analysis of the time and memory costs of algorithms, and with the basics of dynamic programming. The classic text on algorithms is offered by Cormen et al. (2009); for an introduction to the theory of computation, see Arora and Barak (2009) and Sipser (2012).

How to use this book

After the introduction, the textbook is organized into four main units:

Learning. This section builds up a set of machine learning tools that will be used throughout the other sections. Because the focus is on machine learning, the text representations and linguistic phenomena are mostly simple: "bag-of-words" text classification is treated as a model example. Chapter 4 describes some of the more linguistically interesting applications of word-based text analysis.

Sequences and trees. This section introduces the treatment of language as a structured phenomena. It describes sequence and tree representations and the algorithms that they facilitate, as well as the limitations that these representations impose. Chapter 9 introduces finite state automata and briefly overviews a context-free account of English syntax.

Meaning. This section takes a broad view of efforts to represent and compute meaning from text, ranging from formal logic to neural word embeddings. It also includes two topics that are closely related to semantics: resolution of ambiguous references, and analysis of multi-sentence discourse structure.

Applications. The final section offers chapter-length treatments on three of the most prominent applications of natural language processing: information extraction, machine

translation, and text generation. Each of these applications merits a textbook length treatment of its own (Koehn, 2009; Grishman, 2012; Reiter and Dale, 2000); the chapters here explain some of the most well known systems using the formalisms and methods built up earlier in the book, while introducing methods such as neural attention.

Each chapter contains some advanced material, which is marked with an asterisk. This material can be safely omitted without causing misunderstandings later on. But even without these advanced sections, the text is too long for a single semester course, so instructors will have to pick and choose among the chapters.

Chapters 1-3 provide building blocks that will be used throughout the book, and chapter 4 describes some critical aspects of the practice of language technology. Language models (chapter 6), sequence labeling (chapter 7), and parsing (chapter 10 and 11) are canonical topics in natural language processing, and distributed word embeddings (chapter 14) have become ubiquitous. Of the applications, machine translation (chapter 18) is the best choice: it is more cohesive than information extraction, and more mature than text generation. Many students will benefit from the review of probability in Appendix A.

- A course focusing on machine learning should add the chapter on unsupervised learning (chapter 5). The chapters on predicate-argument semantics (chapter 13), reference resolution (chapter 15), and text generation (chapter 19) are particularly influenced by recent progress in machine learning, including deep neural networks and learning to search.
- A course with a more linguistic orientation should add the chapters on applications of sequence labeling (chapter 8), formal language theory (chapter 9), semantics (chapter 12 and 13), and discourse (chapter 16).
- For a course with a more applied focus, I recommend the chapters on applications of sequence labeling (chapter 8), predicate-argument semantics (chapter 13), information extraction (chapter 17), and text generation (chapter 19).

Acknowledgments

Several colleagues, students, and friends read early drafts of chapters in their areas of expertise, including Yoav Artzi, Kevin Duh, Heng Ji, Jessy Li, Brendan O'Connor, Yuval Pinter, Shawn Ling Ramirez, Nathan Schneider, Pamela Shapiro, Noah A. Smith, Sandeep Soni, and Luke Zettlemoyer. I also thank the anonymous reviewers, particularly reviewer 4, who provided detailed line-by-line edits and suggestions. The text benefited from high-level discussions with my editor Marie Lufkin Lee, as well as Kevin Murphy, Shawn Ling Ramirez, and Bonnie Webber. In addition, there are many students, colleagues, friends, and family who found mistakes in early drafts, or who recommended key references.

These include: Parminder Bhatia, Kimberly Caras, Jiahao Cai, Justin Chen, Rodolfo Delmonte, Murtaza Dhuliawala, Yantao Du, Barbara Eisenstein, Luiz C. F. Ribeiro, Chris Gu, Joshua Killingsworth, Jonathan May, Taha Merghani, Gus Monod, Raghavendra Murali, Nidish Nair, Brendan O'Connor, Dan Oneata, Brandon Peck, Yuval Pinter, Nathan Schneider, Jianhao Shen, Zhewei Sun, Rubin Tsui, Ashwin Cunnapakkam Vinjimur, Denny Vrandečić, William Yang Wang, Clay Washington, Ishan Waykul, Aobo Yang, Xavier Yao, Yuyu Zhang, and several anonymous commenters. Clay Washington tested some of the programming exercises, and Varun Gupta tested some of the written exercises. Thanks to Kelvin Xu for sharing a high-resolution version of Figure 19.3.

Most of the book was written while I was at Georgia Tech's School of Interactive Computing. I thank the School for its support of this project, and I thank my colleagues there for their help and support at the beginning of my faculty career. I also thank (and apologize to) the many students in Georgia Tech's CS 4650 and 7650 who suffered through early versions of the text. The book is dedicated to my parents.

Jacob Eisenstein. Draft of November 13, 2018.

Notation

As a general rule, words, word counts, and other types of observations are indicated with Roman letters (a, b, c); parameters are indicated with Greek letters (α, β, θ). Vectors are indicated with bold script for both random variables $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$. Other useful notations are indicated in the table below.

Basics	
$\exp x$	the base-2 exponent, 2^x
$\log x$	the base-2 logarithm, $\log_2 x$
$\{x_n\}_{n=1}^N$	the set $\{x_1, x_2, \ldots, x_N\}$
x_i^j	x_i raised to the power j
$x_i^{(j)}$	indexing by both i and j

Linear algebra	
$\boldsymbol{x}^{(i)}$	a column vector of feature counts for instance i, often word counts
$\boldsymbol{x}_{j:k}$	elements j through k (inclusive) of a vector $\boldsymbol{x}$
$[\boldsymbol{x}; \boldsymbol{y}]$	vertical concatenation of two column vectors
$[\boldsymbol{x}, \boldsymbol{y}]$	horizontal concatenation of two column vectors
$\boldsymbol{e}_n$	a "one-hot" vector with a value of 1 at position n, and zero everywhere else
$\boldsymbol{\theta}^\top$	the transpose of a column vector $\boldsymbol{\theta}$
$\boldsymbol{\theta} \cdot \boldsymbol{x}^{(i)}$	the dot product $\sum_{j=1}^N \theta_j \times x_j^{(i)}$
$\mathbf{X}$	a matrix
$x_{i,j}$	row i, column j of matrix $\mathbf{X}$
Diag($\boldsymbol{x}$)	a matrix with $\boldsymbol{x}$ on the diagonal, e.g., $\begin{pmatrix} x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & x_3 \end{pmatrix}$
$\mathbf{X}^{-1}$	the inverse of matrix $\mathbf{X}$

Text datasets	
w_m	word token at position m
N	number of training instances
M	length of a sequence (of words or tags)
V	number of words in vocabulary
$y^{(i)}$	the true label for instance i
$\hat{y}$	a predicted label
$\mathcal{Y}$	the set of all possible labels
K	number of possible labels $K = \lvert\mathcal{Y}\rvert$
$\square$	the start token
$\blacksquare$	the stop token
$\boldsymbol{y}^{(i)}$	a structured label for instance i, such as a tag sequence
$\mathcal{Y}(\boldsymbol{w})$	the set of possible labelings for the word sequence $\boldsymbol{w}$
$\lozenge$	the start tag
$\blacklozenge$	the stop tag

Probabilities	
$\Pr(A)$	probability of event A
$\Pr(A \mid B)$	probability of event A, conditioned on event B
$\mathrm{p}_B(b)$	the marginal probability of random variable B taking value b; written $\mathrm{p}(b)$ when the choice of random variable is clear from context
$\mathrm{p}_{B\mid A}(b \mid a)$	the probability of random variable B taking value b, conditioned on A taking value a; written $\mathrm{p}(b \mid a)$ when clear from context
$A \sim p$	the random variable A is distributed according to distribution p. For example, $X \sim \mathcal{N}(0, 1)$ states that the random variable X is drawn from a normal distribution with zero mean and unit variance.
$A \mid B \sim p$	conditioned on the random variable B, A is distributed according to p.[2]

Machine learning	
$\Psi(\boldsymbol{x}^{(i)}, y)$	the score for assigning label y to instance i
$\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$	the feature vector for instance i with label y
$\boldsymbol{\theta}$	a (column) vector of weights
$\ell^{(i)}$	loss on an individual instance i
L	objective function for an entire dataset
$\mathcal{L}$	log-likelihood of a dataset
λ	the amount of regularization

Chapter 1

Introduction

Natural language processing is the set of methods for making human language accessible to computers. In the past decade, natural language processing has become embedded in our daily lives: automatic machine translation is ubiquitous on the web and in social media; text classification keeps our email inboxes from collapsing under a deluge of spam; search engines have moved beyond string matching and network analysis to a high degree of linguistic sophistication; dialog systems provide an increasingly common and effective way to get and share information.

These diverse applications are based on a common set of ideas, drawing on algorithms, linguistics, logic, statistics, and more. The goal of this text is to provide a survey of these foundations. The technical fun starts in the next chapter; the rest of this current chapter situates natural language processing with respect to other intellectual disciplines, identifies some high-level themes in contemporary natural language processing, and advises the reader on how best to approach the subject.

1.1 Natural language processing and its neighbors

Natural language processing draws on many other intellectual traditions, from formal linguistics to statistical physics. This section briefly situates natural language processing with respect to some of its closest neighbors.

Computational Linguistics Most of the meetings and journals that host natural language processing research bear the name "computational linguistics", and the terms may be thought of as essentially synonymous. But while there is substantial overlap, there is an important difference in focus. In linguistics, language is the object of study. Computational methods may be brought to bear, just as in scientific disciplines like computational biology and computational astronomy, but they play only a supporting role. In contrast,

natural language processing is focused on the design and analysis of computational algorithms and representations for processing natural human language. The goal of natural language processing is to provide new computational capabilities around human language: for example, extracting information from texts, translating between languages, answering questions, holding a conversation, taking instructions, and so on. Fundamental linguistic insights may be crucial for accomplishing these tasks, but success is ultimately measured by whether and how well the job gets done.

Machine Learning Contemporary approaches to natural language processing rely heavily on machine learning, which makes it possible to build complex computer programs from examples. Machine learning provides an array of general techniques for tasks like converting a sequence of discrete tokens in one vocabulary to a sequence of discrete tokens in another vocabulary — a generalization of what one might informally call "translation." Much of today's natural language processing research can be thought of as applied machine learning. However, natural language processing has characteristics that distinguish it from many of machine learning's other application domains.

- Unlike images or audio, text data is fundamentally discrete, with meaning created by combinatorial arrangements of symbolic units. This is particularly consequential for applications in which text is the output, such as translation and summarization, because it is not possible to gradually approach an optimal solution.
- Although the set of words is discrete, new words are always being created. Furthermore, the distribution over words (and other linguistic elements) resembles that of a **power law**[1] (Zipf, 1949): there will be a few words that are very frequent, and a long tail of words that are rare. A consequence is that natural language processing algorithms must be especially robust to observations that do not occur in the training data.
- Language is **compositional**: units such as words can combine to create phrases, which can combine by the very same principles to create larger phrases. For example, a **noun phrase** can be created by combining a smaller noun phrase with a **prepositional phrase**, as in *the whiteness of the whale*. The prepositional phrase is created by combining a preposition (in this case, *of*) with another noun phrase (*the whale*). In this way, it is possible to create arbitrarily long phrases, such as,

  (1.1) ...huge globular pieces of the whale of the bigness of a human head.[2]

  The meaning of such a phrase must be analyzed in accord with the underlying hierarchical structure. In this case, *huge globular pieces of the whale* acts as a single noun

[1] Throughout the text, **boldface** will be used to indicate keywords that appear in the index.
[2] Throughout the text, this notation will be used to introduce linguistic examples.

phrase, which is conjoined with the prepositional phrase *of the bigness of a human head*. The interpretation would be different if instead, *huge globular pieces* were conjoined with the prepositional phrase *of the whale of the bigness of a human head* — implying a disappointingly small whale. Even though text appears as a sequence, machine learning methods must account for its implicit recursive structure.

Artificial Intelligence The goal of artificial intelligence is to build software and robots with the same range of abilities as humans (Russell and Norvig, 2009). Natural language processing is relevant to this goal in several ways. On the most basic level, the capacity for language is one of the central features of human intelligence, and is therefore a prerequisite for artificial intelligence.[3] Second, much of artificial intelligence research is dedicated to the development of systems that can reason from premises to a conclusion, but such algorithms are only as good as what they know (Dreyfus, 1992). Natural language processing is a potential solution to the "knowledge bottleneck", by acquiring knowledge from texts, and perhaps also from conversations. This idea goes all the way back to Turing's 1949 paper *Computing Machinery and Intelligence*, which proposed the **Turing test** for determining whether artificial intelligence had been achieved (Turing, 2009).

Conversely, reasoning is sometimes essential for basic tasks of language processing, such as resolving a pronoun. **Winograd schemas** are examples in which a single word changes the likely referent of a pronoun, in a way that seems to require knowledge and reasoning to decode (Levesque et al., 2011). For example,

(1.2) The trophy doesn't fit into the brown suitcase because **it** is too [small/large].

When the final word is *small*, then the pronoun *it* refers to the suitcase; when the final word is *large*, then *it* refers to the trophy. Solving this example requires spatial reasoning; other schemas require reasoning about actions and their effects, emotions and intentions, and social conventions.

Such examples demonstrate that natural language understanding cannot be achieved in isolation from knowledge and reasoning. Yet the history of artificial intelligence has been one of increasing specialization: with the growing volume of research in subdisciplines such as natural language processing, machine learning, and computer vision, it is

[3]This view is shared by some, but not all, prominent researchers in artificial intelligence. Michael Jordan, a specialist in machine learning, has said that if he had a billion dollars to spend on any large research project, he would spend it on natural language processing (`https://www.reddit.com/r/MachineLearning/comments/2fxi6v/ama_michael_i_jordan/`). On the other hand, in a public discussion about the future of artificial intelligence in February 2018, computer vision researcher Yann Lecun argued that despite its many practical applications, language is perhaps "number 300" in the priority list for artificial intelligence research, and that it would be a great achievement if AI could attain the capabilities of an orangutan, which do not include language (`http://www.abigailsee.com/2018/02/21/deep-learning-structure-and-innate-priors.html`).

difficult for anyone to maintain expertise across the entire field. Still, recent work has demonstrated interesting connections between natural language processing and other areas of AI, including computer vision (e.g., Antol et al., 2015) and game playing (e.g., Branavan et al., 2009). The dominance of machine learning throughout artificial intelligence has led to a broad consensus on representations such as graphical models and computation graphs, and on algorithms such as backpropagation and combinatorial optimization. Many of the algorithms and representations covered in this text are part of this consensus.

Computer Science The discrete and recursive nature of natural language invites the application of theoretical ideas from computer science. Linguists such as Chomsky and Montague have shown how formal language theory can help to explain the syntax and semantics of natural language. Theoretical models such as finite-state and pushdown automata are the basis for many practical natural language processing systems. Algorithms for searching the combinatorial space of analyses of natural language utterances can be analyzed in terms of their computational complexity, and theoretically motivated approximations can sometimes be applied.

The study of computer systems is also relevant to natural language processing. Large datasets of unlabeled text can be processed more quickly by parallelization techniques like MapReduce (Dean and Ghemawat, 2008; Lin and Dyer, 2010); high-volume data sources such as social media can be summarized efficiently by approximate streaming and sketching techniques (Goyal et al., 2009). When deep neural networks are implemented in production systems, it is possible to eke out speed gains using techniques such as reduced-precision arithmetic (Wu et al., 2016). Many classical natural language processing algorithms are not naturally suited to graphics processing unit (GPU) parallelization, suggesting directions for further research at the intersection of natural language processing and computing hardware (Yi et al., 2011).

Speech Processing Natural language is often communicated in spoken form, and speech recognition is the task of converting an audio signal to text. From one perspective, this is a signal processing problem, which might be viewed as a preprocessing step before natural language processing can be applied. However, context plays a critical role in speech recognition by human listeners: knowledge of the surrounding words influences perception and helps to correct for noise (Miller et al., 1951). For this reason, speech recognition is often integrated with text analysis, particularly with statistical **language models**, which quantify the probability of a sequence of text (see chapter 6). Beyond speech recognition, the broader field of speech processing includes the study of speech-based dialogue systems, which are briefly discussed in chapter 19. Historically, speech processing has often been pursued in electrical engineering departments, while natural language processing

has been the purview of computer scientists. For this reason, the extent of interaction between these two disciplines is less than it might otherwise be.

Ethics As machine learning and artificial intelligence become increasingly ubiquitous, it is crucial to understand how their benefits, costs, and risks are distributed across different kinds of people. Natural language processing raises some particularly salient issues around **ethics, fairness, and accountability**:

Access. Who is natural language processing designed to serve? For example, whose language is translated *from*, and whose language is translated *to*?

Bias. Does language technology learn to replicate social biases from text corpora, and does it reinforce these biases as seemingly objective computational conclusions?

Labor. Whose text and speech comprise the datasets that power natural language processing, and who performs the annotations? Are the benefits of this technology shared with all the people whose work makes it possible?

Privacy and internet freedom. What is the impact of large-scale text processing on the right to free and private communication? What is the potential role of natural language processing in regimes of censorship or surveillance?

This text lightly touches on issues related to fairness and bias in § 14.6.3 and § 18.1.1, but these issues are worthy of a book of their own. For more from within the field of computational linguistics, see the papers from the annual workshop on Ethics in Natural Language Processing (Hovy et al., 2017; Alfano et al., 2018). For an outside perspective on ethical issues relating to data science at large, see boyd and Crawford (2012).

Others Natural language processing plays a significant role in emerging interdisciplinary fields like **computational social science** and the **digital humanities**. Text classification (chapter 4), clustering (chapter 5), and information extraction (chapter 17) are particularly useful tools; another is **probabilistic topic models** (Blei, 2012), which are not covered in this text. **Information retrieval** (Manning et al., 2008) makes use of similar tools, and conversely, techniques such as latent semantic analysis (§ 14.3) have roots in information retrieval. **Text mining** is sometimes used to refer to the application of data mining techniques, especially classification and clustering, to text. While there is no clear distinction between text mining and natural language processing (nor between data mining and machine learning), text mining is typically less concerned with linguistic structure, and more interested in fast, scalable algorithms.

1.2 Three themes in natural language processing

Natural language processing covers a diverse range of tasks, methods, and linguistic phenomena. But despite the apparent incommensurability between, say, the summarization of scientific articles (§ 16.3.4) and the identification of suffix patterns in Spanish verbs (§ 9.1.4), some general themes emerge. The remainder of the introduction focuses on these themes, which will recur in various forms through the text. Each theme can be expressed as an opposition between two extreme viewpoints on how to process natural language. The methods discussed in the text can usually be placed somewhere on the continuum between these two extremes.

1.2.1 Learning and knowledge

A recurring topic of debate is the relative importance of machine learning and linguistic knowledge. On one extreme, advocates of "natural language processing from scratch" (Collobert et al., 2011) propose to use machine learning to train end-to-end systems that transmute raw text into any desired output structure: e.g., a summary, database, or translation. On the other extreme, the core work of natural language processing is sometimes taken to be transforming text into a stack of general-purpose linguistic structures: from subword units called **morphemes**, to word-level **parts-of-speech**, to tree-structured representations of grammar, and beyond, to logic-based representations of meaning. In theory, these general-purpose structures should then be able to support any desired application.

The end-to-end approach has been buoyed by recent results in computer vision and speech recognition, in which advances in machine learning have swept away expert-engineered representations based on the fundamentals of optics and phonology (Krizhevsky et al., 2012; Graves and Jaitly, 2014). But while machine learning is an element of nearly every contemporary approach to natural language processing, linguistic representations such as syntax trees have not yet gone the way of the visual edge detector or the auditory triphone. Linguists have argued for the existence of a "language faculty" in all human beings, which encodes a set of abstractions specially designed to facilitate the understanding and production of language. The argument for the existence of such a language faculty is based on the observation that children learn language faster and from fewer examples than would be possible if language was learned from experience alone.[4] From a practical standpoint, linguistic structure seems to be particularly important in scenarios where training data is limited.

There are a number of ways in which knowledge and learning can be combined in natural language processing. Many supervised learning systems make use of carefully engineered **features**, which transform the data into a representation that can facilitate

[4] *The Language Instinct* (Pinker, 2003) articulates these arguments in an engaging and popular style. For arguments against the innateness of language, see Elman et al. (1998).

learning. For example, in a task like search, it may be useful to identify each word's **stem**, so that a system can more easily generalize across related terms such as *whale*, *whales*, *whalers*, and *whaling*. (This issue is relatively benign in English, as compared to the many other languages which include much more elaborate systems of prefixed and suffixes.) Such features could be obtained from a hand-crafted resource, like a dictionary that maps each word to a single root form. Alternatively, features can be obtained from the output of a general-purpose language processing system, such as a parser or part-of-speech tagger, which may itself be built on supervised machine learning.

Another synthesis of learning and knowledge is in model structure: building machine learning models whose architectures are inspired by linguistic theories. For example, the organization of sentences is often described as **compositional**, with meaning of larger units gradually constructed from the meaning of their smaller constituents. This idea can be built into the architecture of a deep neural network, which is then trained using contemporary deep learning techniques (Dyer et al., 2016).

The debate about the relative importance of machine learning and linguistic knowledge sometimes becomes heated. No machine learning specialist likes to be told that their engineering methodology is unscientific alchemy;[5] nor does a linguist want to hear that the search for general linguistic principles and structures has been made irrelevant by big data. Yet there is clearly room for both types of research: we need to know how far we can go with end-to-end learning alone, while at the same time, we continue the search for linguistic representations that generalize across applications, scenarios, and languages. For more on the history of this debate, see Church (2011); for an optimistic view of the potential symbiosis between computational linguistics and deep learning, see Manning (2015).

1.2.2 Search and learning

Many natural language processing problems can be written mathematically in the form of optimization,[6]

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{x})}{\operatorname{argmax}} \Psi(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}), \quad [1.1]$$

where,

- $\boldsymbol{x}$ is the input, which is an element of a set $\mathcal{X}$;
- $\boldsymbol{y}$ is the output, which is an element of a set $\mathcal{Y}(\boldsymbol{x})$;

[5] Ali Rahimi argued that much of deep learning research was similar to "alchemy" in a presentation at the 2017 conference on Neural Information Processing Systems. He was advocating for more learning theory, not more linguistics.

[6] Throughout this text, equations will be numbered by square brackets, and linguistic examples will be numbered by parentheses.

- Ψ is a scoring function (also called the **model**), which maps from the set $\mathcal{X} \times \mathcal{Y}$ to the real numbers;
- $\boldsymbol{\theta}$ is a vector of parameters for Ψ;
- $\hat{\boldsymbol{y}}$ is the predicted output, which is chosen to maximize the scoring function.

This basic structure can be applied to a huge range of problems. For example, the input $\boldsymbol{x}$ might be a social media post, and the output $\boldsymbol{y}$ might be a labeling of the emotional sentiment expressed by the author (chapter 4); or $\boldsymbol{x}$ could be a sentence in French, and the output $\boldsymbol{y}$ could be a sentence in Tamil (chapter 18); or $\boldsymbol{x}$ might be a sentence in English, and $\boldsymbol{y}$ might be a representation of the syntactic structure of the sentence (chapter 10); or $\boldsymbol{x}$ might be a news article and $\boldsymbol{y}$ might be a structured record of the events that the article describes (chapter 17).

This formulation reflects an implicit decision that language processing algorithms will have two distinct modules:

Search. The search module is responsible for computing the $\max$ of the function Ψ. In other words, it finds the output $\hat{\boldsymbol{y}}$ that gets the best score with respect to the input $\boldsymbol{x}$. This is easy when the search space $\mathcal{Y}(\boldsymbol{x})$ is small enough to enumerate, or when the scoring function Ψ has a convenient decomposition into parts. In many cases, we will want to work with scoring functions that do not have these properties, motivating the use of more sophisticated search algorithms, such as bottom-up dynamic programming (§ 10.1) and beam search (§ 11.3.1). Because the outputs are usually discrete in language processing problems, search often relies on the machinery of **combinatorial optimization**.

Learning. The learning module is responsible for finding the parameters $\boldsymbol{\theta}$. This is typically (but not always) done by processing a large dataset of labeled examples, $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$. Like search, learning is also approached through the framework of optimization, as we will see in chapter 2. Because the parameters are usually continuous, learning algorithms generally rely on **numerical optimization** to identify vectors of real-valued parameters that optimize some function of the model and the labeled data. Some basic principles of numerical optimization are reviewed in Appendix B.

The division of natural language processing into separate modules for search and learning makes it possible to reuse generic algorithms across many tasks and models. Much of the work of natural language processing can be focused on the design of the model Ψ — identifying and formalizing the linguistic phenomena that are relevant to the task at hand — while reaping the benefits of decades of progress in search, optimization, and learning. This textbook will describe several classes of scoring functions, and the corresponding algorithms for search and learning.

When a model is capable of making subtle linguistic distinctions, it is said to be *expressive*. Expressiveness is often traded off against efficiency of search and learning. For example, a word-to-word translation model makes search and learning easy, but it is not expressive enough to distinguish good translations from bad ones. Many of the most important problems in natural language processing seem to require expressive models, in which the complexity of search grows exponentially with the size of the input. In these models, exact search is usually impossible. Intractability threatens the neat modular decomposition between search and learning: if search requires a set of heuristic approximations, then it may be advantageous to learn a model that performs well under these specific heuristics. This has motivated some researchers to take a more integrated approach to search and learning, as briefly mentioned in chapters 11 and 15.

1.2.3 Relational, compositional, and distributional perspectives

Any element of language — a word, a phrase, a sentence, or even a sound — can be described from at least three perspectives. Consider the word *journalist*. A *journalist* is a subcategory of a *profession*, and an *anchorwoman* is a subcategory of *journalist*; furthermore, a *journalist* performs *journalism*, which is often, but not always, a subcategory of *writing*. This relational perspective on meaning is the basis for semantic **ontologies** such as **WordNet** (Fellbaum, 2010), which enumerate the relations that hold between words and other elementary semantic units. The power of the relational perspective is illustrated by the following example:

(1.3) Umashanthi interviewed Ana. She works for the college newspaper.

Who works for the college newspaper? The word *journalist*, while not stated in the example, implicitly links the *interview* to the *newspaper*, making *Umashanthi* the most likely referent for the pronoun. (A general discussion of how to resolve pronouns is found in chapter 15.)

Yet despite the inferential power of the relational perspective, it is not easy to formalize computationally. Exactly which elements are to be related? Are *journalists* and *reporters* distinct, or should we group them into a single unit? Is the kind of *interview* performed by a journalist the same as the kind that one undergoes when applying for a job? Ontology designers face many such thorny questions, and the project of ontology design hearkens back to Borges' (1993) *Celestial Emporium of Benevolent Knowledge*, which divides animals into:

> (a) belonging to the emperor; (b) embalmed; (c) tame; (d) suckling pigs; (e) sirens; (f) fabulous; (g) stray dogs; (h) included in the present classification; (i) frenzied; (j) innumerable; (k) drawn with a very fine camelhair brush; (l) et cetera; (m) having just broken the water pitcher; (n) that from a long way off resemble flies.

Difficulties in ontology construction have led some linguists to argue that there is no task-independent way to partition up word meanings (Kilgarriff, 1997).

Some problems are easier. Each member in a group of *journalists* is a *journalist*: the *-s* suffix distinguishes the plural meaning from the singular in most of the nouns in English. Similarly, a *journalist* can be thought of, perhaps colloquially, as someone who produces or works on a *journal*. (Taking this approach even further, the word *journal* derives from the French *jour+nal*, or *day+ly = daily*.) In this way, the meaning of a word is constructed from the constituent parts — the principle of **compositionality**. This principle can be applied to larger units: phrases, sentences, and beyond. Indeed, one of the great strengths of the compositional view of meaning is that it provides a roadmap for understanding entire texts and dialogues through a single analytic lens, grounding out in the smallest parts of individual words.

But alongside *journalists* and *anti-parliamentarians*, there are many words that seem to be linguistic atoms: think, for example, of *whale*, *blubber*, and *Nantucket*. Idiomatic phrases like *kick the bucket* and *shoot the breeze* have meanings that are quite different from the sum of their parts (Sag et al., 2002). Composition is of little help for such words and expressions, but their meanings can be ascertained — or at least approximated — from the contexts in which they appear. Take, for example, *blubber*, which appears in such contexts as:

(1.4) a. The blubber served them as fuel.
b. . . . extracting it from the blubber of the large fish . . .
c. Amongst oily substances, blubber has been employed as a manure.

These contexts form the **distributional properties** of the word *blubber*, and they link it to words which can appear in similar constructions: *fat*, *pelts*, and *barnacles*. This distributional perspective makes it possible to learn about meaning from unlabeled data alone; unlike relational and compositional semantics, no manual annotation or expert knowledge is required. Distributional semantics is thus capable of covering a huge range of linguistic phenomena. However, it lacks precision: *blubber* is similar to *fat* in one sense, to *pelts* in another sense, and to *barnacles* in still another. The question of *why* all these words tend to appear in the same contexts is left unanswered.

The relational, compositional, and distributional perspectives all contribute to our understanding of linguistic meaning, and all three appear to be critical to natural language processing. Yet they are uneasy collaborators, requiring seemingly incompatible representations and algorithmic approaches. This text presents some of the best known and most successful methods for working with each of these representations, but future research may reveal new ways to combine them.

Part I

Learning

Chapter 2

Linear text classification

We begin with the problem of **text classification**: given a text document, assign it a discrete label $y \in \mathcal{Y}$, where $\mathcal{Y}$ is the set of possible labels. Text classification has many applications, from spam filtering to the analysis of electronic health records. This chapter describes some of the most well known and effective algorithms for text classification, from a mathematical perspective that should help you understand what they do and why they work. Text classification is also a building block in more elaborate natural language processing tasks. For readers without a background in machine learning or statistics, the material in this chapter will take more time to digest than most of the subsequent chapters. But this investment will pay off as the mathematical principles behind these basic classification algorithms reappear in other contexts throughout the book.

2.1 The bag of words

To perform text classification, the first question is how to represent each document, or instance. A common approach is to use a column vector of word counts, e.g., $\boldsymbol{x} = [0, 1, 1, 0, 0, 2, 0, 1, 13, 0 \ldots]^\top$, where x_j is the count of word j. The length of $\boldsymbol{x}$ is $V \triangleq |\mathcal{V}|$, where $\mathcal{V}$ is the set of possible words in the vocabulary. In linear classification, the classification decision is based on a weighted sum of individual feature counts, such as word counts.

The object $\boldsymbol{x}$ is a vector, but it is often called a **bag of words**, because it includes only information about the count of each word, and not the order in which the words appear. With the bag of words representation, we are ignoring grammar, sentence boundaries, paragraphs — everything but the words. Yet the bag of words model is surprisingly effective for text classification. If you see the word *whale* in a document, is it fiction or nonfiction? What if you see the word *molybdenum*? For many labeling problems, individual words can be strong predictors.

To predict a label from a bag-of-words, we can assign a score to each word in the vocabulary, measuring the compatibility with the label. For example, for the label FICTION, we might assign a positive score to the word *whale*, and a negative score to the word *molybdenum*. These scores are called **weights**, and they are arranged in a column vector $\boldsymbol{\theta}$.

Suppose that you want a multiclass classifier, where $K \triangleq |\mathcal{Y}| > 2$. For example, you might want to classify news stories about sports, celebrities, music, and business. The goal is to predict a label $\hat{y}$, given the bag of words $\boldsymbol{x}$, using the weights $\boldsymbol{\theta}$. For each label $y \in \mathcal{Y}$, we compute a score $\Psi(\boldsymbol{x}, y)$, which is a scalar measure of the compatibility between the bag-of-words $\boldsymbol{x}$ and the label y. In a linear bag-of-words classifier, this score is the vector inner product between the weights $\boldsymbol{\theta}$ and the output of a **feature function** $\boldsymbol{f}(\boldsymbol{x}, y)$,

$$\Psi(\boldsymbol{x}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) = \sum_j \theta_j f_j(\boldsymbol{x}, y). \quad [2.1]$$

As the notation suggests, $\boldsymbol{f}$ is a function of two arguments, the word counts $\boldsymbol{x}$ and the label y, and it returns a vector output. For example, given arguments $\boldsymbol{x}$ and y, element j of this feature vector might be,

$$f_j(\boldsymbol{x}, y) = \begin{cases} x_{\textit{whale}}, & \text{if } y = \text{FICTION} \\ 0, & \text{otherwise} \end{cases} \quad [2.2]$$

This function returns the count of the word *whale* if the label is FICTION, and it returns zero otherwise. The index j depends on the position of *whale* in the vocabulary, and of FICTION in the set of possible labels. The corresponding weight θ_j then scores the compatibility of the word *whale* with the label FICTION.[1] A positive score means that this word makes the label more likely.

The output of the feature function can be formalized as a vector:

$$\boldsymbol{f}(\boldsymbol{x}, y = 1) = [\boldsymbol{x}; \underbrace{0; 0; \ldots; 0}_{(K-1)\times V}] \quad [2.3]$$

$$\boldsymbol{f}(\boldsymbol{x}, y = 2) = [\underbrace{0; 0; \ldots; 0}_{V}; \boldsymbol{x}; \underbrace{0; 0; \ldots; 0}_{(K-2)\times V}] \quad [2.4]$$

$$\boldsymbol{f}(\boldsymbol{x}, y = K) = [\underbrace{0; 0; \ldots; 0}_{(K-1)\times V}; \boldsymbol{x}], \quad [2.5]$$

where $[\underbrace{0; 0; \ldots; 0}_{(K-1)\times V}]$ is a column vector of $(K-1) \times V$ zeros, and the semicolon indicates vertical concatenation. For each of the K possible labels, the feature function returns a

[1] In practice, both $\boldsymbol{f}$ and $\boldsymbol{\theta}$ may be implemented as a dictionary rather than vectors, so that it is not necessary to explicitly identify j. In such an implementation, the tuple (*whale*, FICTION) acts as a key in both dictionaries; the values in $\boldsymbol{f}$ are feature counts, and the values in $\boldsymbol{\theta}$ are weights.

vector that is mostly zeros, with a column vector of word counts $\boldsymbol{x}$ inserted in a location that depends on the specific label y. This arrangement is shown in Figure 2.1. The notation may seem awkward at first, but it generalizes to an impressive range of learning settings, particularly **structure prediction**, which is the focus of Chapters 7-11.

Given a vector of weights, $\boldsymbol{\theta} \in \mathbb{R}^{VK}$, we can now compute the score $\Psi(\boldsymbol{x}, y)$ by Equation 2.1. This inner product gives a scalar measure of the compatibility of the observation $\boldsymbol{x}$ with label y.[2] For any document $\boldsymbol{x}$, we predict the label $\hat{y}$,

$$\hat{y} = \operatorname*{argmax}_{y \in \mathcal{Y}} \Psi(\boldsymbol{x}, y) \quad [2.6]$$

$$\Psi(\boldsymbol{x}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y). \quad [2.7]$$

This inner product notation gives a clean separation between the *data* ($\boldsymbol{x}$ and y) and the *parameters* ($\boldsymbol{\theta}$).

While vector notation is used for presentation and analysis, in code the weights and feature vector can be implemented as dictionaries. The inner product can then be computed as a loop. In python:

```
def compute_score(x,y,weights):
    total = 0
    for feature,count in feature_function(x,y).items():
        total += weights[feature] * count
    return total
```

This representation is advantageous because it avoids storing and iterating over the many features whose counts are zero.

It is common to add an **offset feature** at the end of the vector of word counts $\boldsymbol{x}$, which is always 1. We then have to also add an extra zero to each of the zero vectors, to make the vector lengths match. This gives the entire feature vector $\boldsymbol{f}(\boldsymbol{x}, y)$ a length of $(V + 1) \times K$. The weight associated with this offset feature can be thought of as a bias for or against each label. For example, if we expect most emails to be spam, then the weight for the offset feature for $y = \text{SPAM}$ should be larger than the weight for the offset feature for $y = \text{NOT-SPAM}$.

Returning to the weights $\boldsymbol{\theta}$, where do they come from? One possibility is to set them by hand. If we wanted to distinguish, say, English from Spanish, we can use English and Spanish dictionaries, and set the weight to one for each word that appears in the

[2] Only $V \times (K - 1)$ features and weights are necessary. By stipulating that $\Psi(\boldsymbol{x}, y = K) = 0$ regardless of $\boldsymbol{x}$, it is possible to implement any classification rule that can be achieved with $V \times K$ features and weights. This is the approach taken in binary classification rules like $y = \text{Sign}(\boldsymbol{\beta} \cdot \boldsymbol{x} + a)$, where $\boldsymbol{\beta}$ is a vector of weights, a is an offset, and the label set is $\mathcal{Y} = \{-1, 1\}$. However, for multiclass classification, it is more concise to write $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ for all $y \in \mathcal{Y}$.

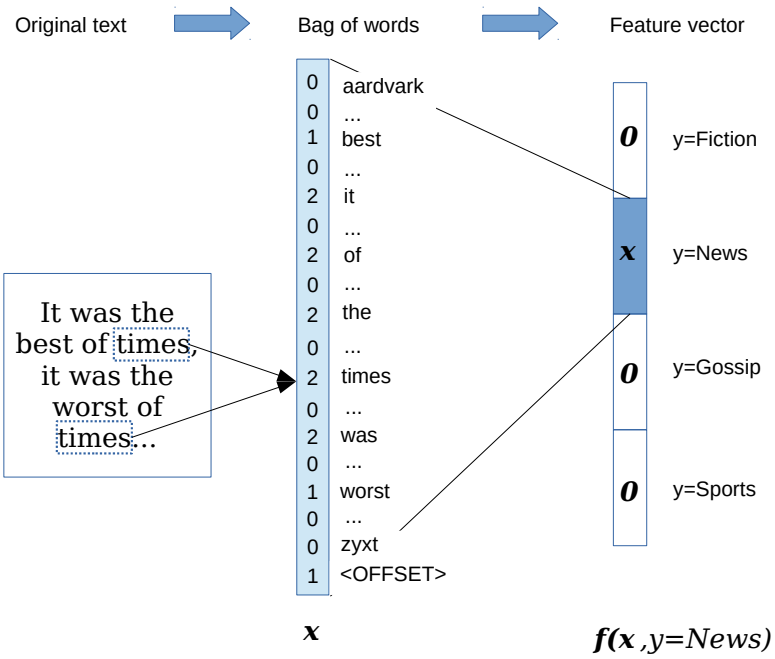


Figure 2.1: The bag-of-words and feature vector representations, for a hypothetical text classification task.

associated dictionary. For example,[3]

$$\theta_{(E,bicycle)} = 1 \qquad \theta_{(S,bicycle)} = 0$$
$$\theta_{(E,bicicleta)} = 0 \qquad \theta_{(S,bicicleta)} = 1$$
$$\theta_{(E,con)} = 1 \qquad \theta_{(S,con)} = 1$$
$$\theta_{(E,ordinateur)} = 0 \qquad \theta_{(S,ordinateur)} = 0.$$

Similarly, if we want to distinguish positive and negative sentiment, we could use positive and negative **sentiment lexicons** (see § 4.1.2), which are defined by social psychologists (Tausczik and Pennebaker, 2010).

But it is usually not easy to set classification weights by hand, due to the large number of words and the difficulty of selecting exact numerical weights. Instead, we will learn the weights from data. Email users manually label messages as SPAM; newspapers label their own articles as BUSINESS or STYLE. Using such **instance labels**, we can automatically acquire weights using **supervised machine learning**. This chapter will discuss several machine learning approaches for classification. The first is based on probability. For a review of probability, consult Appendix A.

[3] In this notation, each tuple (language, word) indexes an element in $\boldsymbol{\theta}$, which remains a vector.

2.2 Naïve Bayes

The **joint probability** of a bag of words $\boldsymbol{x}$ and its true label y is written $\text{p}(\boldsymbol{x}, y)$. Suppose we have a dataset of N labeled instances, $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, which we assume are **independent and identically distributed (IID)** (see § A.3). Then the joint probability of the entire dataset, written $\text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)})$, is equal to $\prod_{i=1}^{N} \text{p}_{X,Y}(\boldsymbol{x}^{(i)}, y^{(i)})$.[4]

What does this have to do with classification? One approach to classification is to set the weights $\boldsymbol{\theta}$ so as to maximize the joint probability of a **training set** of labeled documents. This is known as **maximum likelihood estimation**:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmax}}\, \text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)}; \boldsymbol{\theta}) \quad [2.8]$$

$$= \underset{\boldsymbol{\theta}}{\text{argmax}} \prod_{i=1}^{N} \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \quad [2.9]$$

$$= \underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \quad [2.10]$$

The notation $\text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$ indicates that $\boldsymbol{\theta}$ is a *parameter* of the probability function. The product of probabilities can be replaced by a sum of log-probabilities because the $\log$ function is monotonically increasing over positive arguments, and so the same $\boldsymbol{\theta}$ will maximize both the probability and its logarithm. Working with logarithms is desirable because of numerical stability: on a large dataset, multiplying many probabilities can **underflow** to zero.[5]

The probability $\text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$ is defined through a **generative model** — an idealized random process that has generated the observed data.[6] Algorithm 1 describes the generative model underlying the **Naïve Bayes** classifier, with parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\phi}\}$.

- The first line of this generative model encodes the assumption that the instances are mutually independent: neither the label nor the text of document i affects the label or text of document j.[7] Furthermore, the instances are identically distributed: the

[4] The notation $\text{p}_{X,Y}(\boldsymbol{x}^{(i)}, y^{(i)})$ indicates the joint probability that random variables X and Y take the specific values $\boldsymbol{x}^{(i)}$ and $y^{(i)}$ respectively. The subscript will often be omitted when it is clear from context. For a review of random variables, see Appendix A.

[5] Throughout this text, you may assume all logarithms and exponents are base 2, unless otherwise indicated. Any reasonable base will yield an identical classifier, and base 2 is most convenient for working out examples by hand.

[6] Generative models will be used throughout this text. They explicitly define the assumptions underlying the form of a probability distribution over observed and latent variables. For a readable introduction to generative models in statistics, see Blei (2014).

[7] Can you think of any cases in which this assumption is too strong?

Algorithm 1 Generative process for the Naïve Bayes classification model

for Instance $i \in \{1, 2, \ldots, N\}$ **do**:
  Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$;
  Draw the word counts $\boldsymbol{x}^{(i)} \mid y^{(i)} \sim \text{Multinomial}(\boldsymbol{\phi}_{y^{(i)}})$.

distributions over the label $y^{(i)}$ and the text $\boldsymbol{x}^{(i)}$ (conditioned on $y^{(i)}$) are the same for all instances i. In other words, we make the assumption that every document has the same distribution over labels, and that each document's distribution over words depends only on the label, and not on anything else about the document. We also assume that the documents don't affect each other: if the word *whale* appears in document $i = 7$, that does not make it any more or less likely that it will appear again in document $i = 8$.

- The second line of the generative model states that the random variable $y^{(i)}$ is drawn from a categorical distribution with parameter $\boldsymbol{\mu}$. Categorical distributions are like weighted dice: the column vector $\boldsymbol{\mu} = [\mu_1; \mu_2; \ldots; \mu_K]$ gives the probabilities of each label, so that the probability of drawing label y is equal to μ_y. For example, if $\mathcal{Y} = \{\text{POSITIVE}, \text{NEGATIVE}, \text{NEUTRAL}\}$, we might have $\boldsymbol{\mu} = [0.1; 0.7; 0.2]$. We require $\sum_{y \in \mathcal{Y}} \mu_y = 1$ and $\mu_y \geq 0$, $\forall y \in \mathcal{Y}$: each label's probability is non-negative, and the sum of these probabilities is equal to one. [8]
- The third line describes how the bag-of-words counts $\boldsymbol{x}^{(i)}$ are generated. By writing $\boldsymbol{x}^{(i)} \mid y^{(i)}$, this line indicates that the word counts are conditioned on the label, so that the joint probability is factored using the chain rule,

$$\text{p}_{X,Y}(\boldsymbol{x}^{(i)}, y^{(i)}) = \text{p}_{X|Y}(\boldsymbol{x}^{(i)} \mid y^{(i)}) \times \text{p}_Y(y^{(i)}). \quad [2.11]$$

  The specific distribution $\text{p}_{X|Y}$ is the **multinomial**, which is a probability distribution over vectors of non-negative counts. The probability mass function for this distribution is:

$$\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}) = B(\boldsymbol{x}) \prod_{j=1}^{V} \phi_j^{x_j} \quad [2.12]$$

$$B(\boldsymbol{x}) = \frac{\left(\sum_{j=1}^{V} x_j\right)!}{\prod_{j=1}^{V} (x_j!)}. \quad [2.13]$$

[8] Formally, we require $\boldsymbol{\mu} \in \Delta^{K-1}$, where Δ^{K-1} is the $K-1$ **probability simplex**, the set of all vectors of K nonnegative numbers that sum to one. Because of the sum-to-one constraint, there are $K-1$ degrees of freedom for a vector of size K.

As in the categorical distribution, the parameter ϕ_j can be interpreted as a probability: specifically, the probability that any given token in the document is the word j. The multinomial distribution involves a product over words, with each term in the product equal to the probability ϕ_j, exponentiated by the count x_j. Words that have zero count play no role in this product, because $\phi_j^0 = 1$. The term $B(\boldsymbol{x})$ is called the **multinomial coefficient**. It doesn't depend on $\boldsymbol{\phi}$, and can usually be ignored. Can you see why we need this term at all?[9]

The notation $\text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})$ indicates the conditional probability of word counts $\boldsymbol{x}$ given label y, with parameter $\boldsymbol{\phi}$, which is equal to $\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}_y)$. By specifying the multinomial distribution, we describe the **multinomial Naïve Bayes** classifier. Why "naïve"? Because the multinomial distribution treats each word token independently, conditioned on the class: the probability mass function factorizes across the counts.[10]

2.2.1 Types and tokens

A slight modification to the generative model of Naïve Bayes is shown in Algorithm 2. Instead of generating a vector of counts of **types**, $\boldsymbol{x}$, this model generates a *sequence* of **tokens**, $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$. The distinction between types and tokens is critical: $x_j \in \{0, 1, 2, \ldots, M\}$ is the count of word type j in the vocabulary, e.g., the number of times the word *cannibal* appears; $w_m \in \mathcal{V}$ is the identity of token m in the document, e.g. $w_m =$ *cannibal*.

The probability of the sequence $\boldsymbol{w}$ is a product of categorical probabilities. Algorithm 2 makes a conditional independence assumption: each token $w_m^{(i)}$ is independent of all other tokens $w_{n \neq m}^{(i)}$, conditioned on the label $y^{(i)}$. This is identical to the "naïve" independence assumption implied by the multinomial distribution, and as a result, the optimal parameters for this model are identical to those in multinomial Naïve Bayes. For any instance, the probability assigned by this model is proportional to the probability under multinomial Naïve Bayes. The constant of proportionality is the multinomial coefficient $B(\boldsymbol{x})$. Because $B(\boldsymbol{x}) \geq 1$, the probability for a vector of counts $\boldsymbol{x}$ is at least as large as the probability for a list of words $\boldsymbol{w}$ that induces the same counts: there can be many word sequences that correspond to a single vector of counts. For example, *man bites dog* and *dog bites man* correspond to an identical count vector, $\{\textit{bites} : 1, \textit{dog} : 1, \textit{man} : 1\}$, and $B(\boldsymbol{x})$ is equal to the total number of possible word orderings for count vector $\boldsymbol{x}$.

[9] Technically, a multinomial distribution requires a second parameter, the total number of word counts in $\boldsymbol{x}$. In the bag-of-words representation is equal to the number of words in the document. However, this parameter is irrelevant for classification.

[10] You can plug in any probability distribution to the generative story and it will still be Naïve Bayes, as long as you are making the "naïve" assumption that the features are conditionally independent, given the label. For example, a multivariate Gaussian with diagonal covariance is naïve in exactly the same sense.

Algorithm 2 Alternative generative process for the Naïve Bayes classification model

for Instance $i \in \{1, 2, \ldots, N\}$ **do**:
  Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$;
  for Token $m \in \{1, 2, \ldots, M_i\}$ **do**:
    Draw the token $w_m^{(i)} \mid y^{(i)} \sim \text{Categorical}(\boldsymbol{\phi}_{y^{(i)}})$.

Sometimes it is useful to think of instances as counts of types, $\boldsymbol{x}$; other times, it is better to think of them as sequences of tokens, $\boldsymbol{w}$. If the tokens are generated from a model that assumes conditional independence, then these two views lead to probability models that are identical, except for a scaling factor that does not depend on the label or the parameters.

2.2.2 Prediction

The Naïve Bayes prediction rule is to choose the label y which maximizes $\log \text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi})$:

$$\hat{y} = \underset{y}{\text{argmax}} \log \text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi}) \quad [2.14]$$

$$= \underset{y}{\text{argmax}} \log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) \quad [2.15]$$

Now we can plug in the probability distributions from the generative story.

$$\log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) = \log \left[B(\boldsymbol{x}) \prod_{j=1}^{V} \phi_{y,j}^{x_j} \right] + \log \mu_y \quad [2.16]$$

$$= \log B(\boldsymbol{x}) + \sum_{j=1}^{V} x_j \log \phi_{y,j} + \log \mu_y \quad [2.17]$$

$$= \log B(\boldsymbol{x}) + \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y), \quad [2.18]$$

where

$$\boldsymbol{\theta} = [\boldsymbol{\theta}^{(1)}; \boldsymbol{\theta}^{(2)}; \ldots; \boldsymbol{\theta}^{(K)}] \quad [2.19]$$

$$\boldsymbol{\theta}^{(y)} = [\log \phi_{y,1}; \log \phi_{y,2}; \ldots; \log \phi_{y,V}; \log \mu_y] \quad [2.20]$$

The feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ is a vector of V word counts and an offset, padded by zeros for the labels not equal to y (see Equations 2.3-2.5, and Figure 2.1). This construction ensures that the inner product $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ only activates the features whose weights are in $\boldsymbol{\theta}^{(y)}$. These features and weights are all we need to compute the joint log-probability $\log \text{p}(\boldsymbol{x}, y)$ for each y. This is a key point: through this notation, we have converted the problem of computing the log-likelihood for a document-label pair $(\boldsymbol{x}, y)$ into the computation of a vector inner product.

2.2.3 Estimation

The parameters of the categorical and multinomial distributions have a simple interpretation: they are vectors of expected frequencies for each possible event. Based on this interpretation, it is tempting to set the parameters empirically,

$$\phi_{y,j} = \frac{\text{count}(y, j)}{\sum_{j'=1}^{V} \text{count}(y, j')} = \frac{\sum_{i:y^{(i)}=y} x_j^{(i)}}{\sum_{j'=1}^{V} \sum_{i:y^{(i)}=y} x_{j'}^{(i)}}, \quad [2.21]$$

where $\text{count}(y, j)$ refers to the count of word j in documents with label y.

Equation 2.21 defines the **relative frequency estimate** for $\boldsymbol{\phi}$. It can be justified as a **maximum likelihood estimate**: the estimate that maximizes the probability $\text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)}; \boldsymbol{\theta})$. Based on the generative model in Algorithm 1, the log-likelihood is,

$$\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\mu}) = \sum_{i=1}^{N} \log \text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) + \log \text{p}_{\text{cat}}(y^{(i)}; \boldsymbol{\mu}), \quad [2.22]$$

which is now written as a function $\mathcal{L}$ of the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$. Let's continue to focus on the parameters $\boldsymbol{\phi}$. Since $\text{p}(y)$ is constant with respect to $\boldsymbol{\phi}$, we can drop it:

$$\mathcal{L}(\boldsymbol{\phi}) = \sum_{i=1}^{N} \log \text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) = \sum_{i=1}^{N} \log B(\boldsymbol{x}^{(i)}) + \sum_{j=1}^{V} x_j^{(i)} \log \phi_{y^{(i)},j}, \quad [2.23]$$

where $B(\boldsymbol{x}^{(i)})$ is constant with respect to $\boldsymbol{\phi}$.

Maximum-likelihood estimation chooses $\boldsymbol{\phi}$ to maximize the log-likelihood $\mathcal{L}$. However, the solution must obey the following constraints:

$$\sum_{j=1}^{V} \phi_{y,j} = 1 \quad \forall y \quad [2.24]$$

These constraints can be incorporated by adding a set of Lagrange multipliers to the objective (see Appendix B for more details). To solve for each $\boldsymbol{\theta}_y$, we maximize the Lagrangian,

$$\ell(\boldsymbol{\phi}_y) = \sum_{i:y^{(i)}=y} \sum_{j=1}^{V} x_j^{(i)} \log \phi_{y,j} - \lambda(\sum_{j=1}^{V} \phi_{y,j} - 1). \quad [2.25]$$

Differentiating with respect to the parameter $\phi_{y,j}$ yields,

$$\frac{\partial \ell(\boldsymbol{\phi}_y)}{\partial \phi_{y,j}} = \sum_{i:y^{(i)}=y} x_j^{(i)} / \phi_{y,j} - \lambda. \quad [2.26]$$

The solution is obtained by setting each element in this vector of derivatives equal to zero,

$$\lambda\phi_{y,j} = \sum_{i:y^{(i)}=y} x_j^{(i)} \quad [2.27]$$

$$\phi_{y,j} \propto \sum_{i:y^{(i)}=y} x_j^{(i)} = \sum_{i=1}^{N} \delta\left(y^{(i)} = y\right) x_j^{(i)} = \text{count}(y, j), \quad [2.28]$$

where $\delta\left(y^{(i)} = y\right)$ is a **delta function**, also sometimes called an indicator function, which returns one if $y^{(i)} = y$. The symbol $\propto$ indicates that $\phi_{y,j}$ is **proportional to** the right-hand side of the equation.

Equation 2.28 shows three different notations for the same thing: a sum over the word counts for all documents i such that the label $y^{(i)} = y$. This gives a solution for each $\boldsymbol{\phi}_y$ up to a constant of proportionality. Now recall the constraint $\sum_{j=1}^{V} \boldsymbol{\phi}_{y,j} = 1$, which arises because $\boldsymbol{\phi}_y$ represents a vector of probabilities for each word in the vocabulary. This constraint leads to an exact solution, which does not depend on λ:

$$\phi_{y,j} = \frac{\text{count}(y, j)}{\sum_{j'=1}^{V} \text{count}(y, j')}. \quad [2.29]$$

This is equal to the relative frequency estimator from Equation 2.21. A similar derivation gives $\mu_y \propto \sum_{i=1}^{N} \delta\left(y^{(i)} = y\right)$.

2.2.4 Smoothing

With text data, there are likely to be pairs of labels and words that never appear in the training set, leaving $\phi_{y,j} = 0$. For example, the word *molybdenum* may have never yet appeared in a work of fiction. But choosing a value of $\phi_{\text{FICTION},\textit{molybdenum}} = 0$ would allow this single feature to completely veto a label, since $\text{p}(\text{FICTION} \mid \boldsymbol{x}) = 0$ if $\boldsymbol{x}_{\textit{molybdenum}} > 0$.

This is undesirable, because it imposes high **variance**: depending on what data happens to be in the training set, we could get vastly different classification rules. One solution is to **smooth** the probabilities, by adding a "pseudocount" of α to each count, and then normalizing.

$$\phi_{y,j} = \frac{\alpha + \text{count}(y, j)}{V\alpha + \sum_{j'=1}^{V} \text{count}(y, j')} \quad [2.30]$$

This is called **Laplace smoothing**.[11] The pseudocount α is a **hyperparameter**, because it controls the form of the log-likelihood function, which in turn drives the estimation of $\boldsymbol{\phi}$.

[11] Laplace smoothing has a Bayesian justification, in which the generative model is extended to include $\boldsymbol{\phi}$ as a random variable. The resulting distribution over $\boldsymbol{\phi}$ depends on both the data ($\boldsymbol{x}$ and y) and the **prior probability** $\text{p}(\boldsymbol{\phi}; \alpha)$. The corresponding estimate of $\boldsymbol{\phi}$ is called **maximum a posteriori**, or MAP. This is in contrast with maximum likelihood, which depends only on the data.

Smoothing reduces variance, but moves us away from the maximum likelihood estimate: it imposes a **bias**. In this case, the bias points towards uniform probabilities. Machine learning theory shows that errors on heldout data can be attributed to the sum of bias and variance (Mohri et al., 2012). In general, techniques for reducing variance often increase the bias, leading to a **bias-variance tradeoff**.

- Unbiased classifiers may **overfit** the training data, yielding poor performance on unseen data.
- But if the smoothing is too large, the resulting classifier can **underfit** instead. In the limit of $\alpha \rightarrow \infty$, there is zero variance: you get the same classifier, regardless of the data. However, the bias is likely to be large.

Similar issues arise throughout machine learning. Later in this chapter we will encounter **regularization**, which controls the bias-variance tradeoff for logistic regression and large-margin classifiers (§ 2.5.1); § 3.3.2 describes techniques for controlling variance in deep learning; chapter 6 describes more elaborate methods for smoothing empirical probabilities.

2.2.5 Setting hyperparameters

Returning to Naïve Bayes, how should we choose the best value of hyperparameters like α? Maximum likelihood will not work: the maximum likelihood estimate of α on the training set will always be $\alpha = 0$. In many cases, what we really want is **accuracy**: the number of correct predictions, divided by the total number of predictions. (Other measures of classification performance are discussed in § 4.4.) As we will see, it is hard to optimize for accuracy directly. But for scalar hyperparameters like α, tuning can be performed by a simple heuristic called **grid search**: try a set of values (e.g., $\alpha \in \{0.001, 0.01, 0.1, 1, 10\}$), compute the accuracy for each value, and choose the setting that maximizes the accuracy.

The goal is to tune α so that the classifier performs well on *unseen* data. For this reason, the data used for hyperparameter tuning should not overlap the training set, where very small values of α will be preferred. Instead, we hold out a **development set** (also called a **tuning set**) for hyperparameter selection. This development set may consist of a small fraction of the labeled data, such as 10%.

We also want to predict the performance of our classifier on unseen data. To do this, we must hold out a separate subset of data, called the **test set**. It is critical that the test set not overlap with either the training or development sets, or else we will overestimate the performance that the classifier will achieve on unlabeled data in the future. The test set should also not be used when making modeling decisions, such as the form of the feature function, the size of the vocabulary, and so on (these decisions are reviewed in chapter 4.) The ideal practice is to use the test set only once — otherwise, the test set is used to guide

the classifier design, and test set accuracy will diverge from accuracy on truly unseen data. Because annotated data is expensive, this ideal can be hard to follow in practice, and many test sets have been used for decades. But in some high-impact applications like machine translation and information extraction, new test sets are released every year.

When only a small amount of labeled data is available, the test set accuracy can be unreliable. K-fold **cross-validation** is one way to cope with this scenario: the labeled data is divided into K folds, and each fold acts as the test set, while training on the other folds. The test set accuracies are then aggregated. In the extreme, each fold is a single data point; this is called **leave-one-out cross-validation**. To perform hyperparameter tuning in the context of cross-validation, another fold can be used for grid search. It is important not to repeatedly evaluate the cross-validated accuracy while making design decisions about the classifier, or you will overstate the accuracy on truly unseen data.

2.3 Discriminative learning

Naïve Bayes is easy to work with: the weights can be estimated in closed form, and the probabilistic interpretation makes it relatively easy to extend. However, the assumption that features are independent can seriously limit its accuracy. Thus far, we have defined the feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ so that it corresponds to bag-of-words features: one feature per word in the vocabulary. In natural language, bag-of-words features violate the assumption of conditional independence — for example, the probability that a document will contain the word *naïve* is surely higher given that it also contains the word *Bayes* — but this violation is relatively mild.

However, good performance on text classification often requires features that are richer than the bag-of-words:

- To better handle out-of-vocabulary terms, we want features that apply to multiple words, such as prefixes and suffixes (e.g., *anti-*, *un-*, *-ing*) and capitalization.
- We also want n**-gram** features that apply to multi-word units: **bigrams** (e.g., *not good*, *not bad*), **trigrams** (e.g., *not so bad*, *lacking any decency*, *never before imagined*), and beyond.

These features flagrantly violate the Naïve Bayes independence assumption. Consider what happens if we add a prefix feature. Under the Naïve Bayes assumption, the joint probability of a word and its prefix are computed with the following approximation:[12]

$$\Pr(\text{word} = \textit{unfit}, \text{prefix} = \textit{un-} \mid y) \approx \Pr(\text{prefix} = \textit{un-} \mid y) \times \Pr(\text{word} = \textit{unfit} \mid y).$$

[12]The notation $\Pr(\cdot)$ refers to the probability of an event, and $\mathrm{p}(\cdot)$ refers to the probability density or mass for a random variable (see Appendix A).

To test the quality of the approximation, we can manipulate the left-hand side by applying the chain rule,

$$\Pr(\text{word} = \textit{unfit}, \text{prefix} = \textit{un-} \mid y) = \Pr(\text{prefix} = \textit{un-} \mid \text{word} = \textit{unfit}, y) \quad [2.31]$$
$$\times \Pr(\text{word} = \textit{unfit} \mid y) \quad [2.32]$$

But $\Pr(\text{prefix} = \textit{un-} \mid \text{word} = \textit{unfit}, y) = 1$, since *un-* is guaranteed to be the prefix for the word *unfit*. Therefore,

$$\Pr(\text{word} = \textit{unfit}, \text{prefix} = \textit{un-} \mid y) = 1 \times \Pr(\text{word} = \textit{unfit} \mid y) \quad [2.33]$$
$$\gg \Pr(\text{prefix} = \textit{un-} \mid y) \times \Pr(\text{word} = \textit{unfit} \mid y), \quad [2.34]$$

because the probability of any given word starting with the prefix *un-* is much less than one. Naïve Bayes will systematically underestimate the true probabilities of conjunctions of positively correlated features. To use such features, we need learning algorithms that do not rely on an independence assumption.

The origin of the Naïve Bayes independence assumption is the learning objective, $\text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)})$, which requires modeling the probability of the observed text. In classification problems, we are always given $\boldsymbol{x}$, and are only interested in predicting the label y. In this setting, modeling the probability of the text $\boldsymbol{x}$ seems like a difficult and unnecessary task. **Discriminative learning** algorithms avoid this task, and focus directly on the problem of predicting y.

2.3.1 Perceptron

In Naïve Bayes, the weights can be interpreted as parameters of a probabilistic model. But this model requires an independence assumption that usually does not hold, and limits our choice of features. Why not forget about probability and learn the weights in an error-driven way? The **perceptron** algorithm, shown in Algorithm 3, is one way to do this.

The algorithm is simple: if you make a mistake, increase the weights for features that are active with the correct label $y^{(i)}$, and decrease the weights for features that are active with the guessed label $\hat{y}$. Perceptron is an **online learning** algorithm, since the classifier weights change after every example. This is different from Naïve Bayes, which is a **batch learning** algorithm: it computes statistics over the entire dataset, and then sets the weights in a single operation. Algorithm 3 is vague about when this online learning procedure terminates. We will return to this issue shortly.

The perceptron algorithm may seem like an unprincipled heuristic: Naïve Bayes has a solid foundation in probability, but the perceptron is just adding and subtracting constants from the weights every time there is a mistake. Will this really work? In fact, there is some nice theory for the perceptron, based on the concept of **linear separability**. Informally, a dataset with binary labels ($y \in \{0, 1\}$) is linearly separable if it is possible to draw a

Algorithm 3 Perceptron learning algorithm

```
procedure PERCEPTRON(x^(1:N), y^(1:N))
    t ← 0
    θ^(0) ← 0
    repeat
        t ← t + 1
        Select an instance i
        ŷ ← argmax_y θ^(t−1) · f(x^(i), y)
        if ŷ ≠ y^(i) then
            θ^(t) ← θ^(t−1) + f(x^(i), y^(i)) − f(x^(i), ŷ)
        else
            θ^(t) ← θ^(t−1)
    until tired
    return θ^(t)
```

hyperplane (a line in many dimensions), such that on each side of the hyperplane, all instances have the same label. This definition can be formalized and extended to multiple labels:

Definition 1 (Linear separability)**.** *The dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$ is linearly separable iff (if and only if) there exists some weight vector $\boldsymbol{\theta}$ and some **margin** ρ such that for every instance $(\boldsymbol{x}^{(i)}, y^{(i)})$, the inner product of $\boldsymbol{\theta}$ and the feature function for the true label, $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$, is at least ρ greater than inner product of $\boldsymbol{\theta}$ and the feature function for every other possible label, $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')$.*

$$\exists \boldsymbol{\theta}, \rho > 0 : \forall (\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{D}, \quad \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \geq \rho + \max_{y' \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.35}$$

Linear separability is important because of the following guarantee: if your data is linearly separable, then the perceptron algorithm will find a separator (Novikoff, 1962).[13] So while the perceptron may seem heuristic, it is guaranteed to succeed, if the learning problem is easy enough.

How useful is this proof? Minsky and Papert (1969) famously proved that the simple logical function of *exclusive-or* is not separable, and that a perceptron is therefore incapable of learning this function. But this is not just an issue for the perceptron: any linear classification algorithm, including Naïve Bayes, will fail on this task. Text classification problems usually involve high dimensional feature spaces, with thousands or millions of

[13] It is also possible to prove an upper bound on the number of training iterations required to find the separator. Proofs like this are part of the field of **machine learning theory** (Mohri et al., 2012).

features. For these problems, it is very likely that the training data is indeed separable. And even if the dataset is not separable, it is still possible to place an upper bound on the number of errors that the perceptron algorithm will make (Freund and Schapire, 1999).

2.3.2 Averaged perceptron

The perceptron iterates over the data repeatedly — until "tired", as described in Algorithm 3. If the data is linearly separable, the perceptron will eventually find a separator, and we can stop once all training instances are classified correctly. But if the data is not linearly separable, the perceptron can *thrash* between two or more weight settings, never converging. In this case, how do we know that we can stop training, and how should we choose the final weights? An effective practical solution is to *average* the perceptron weights across all iterations.

This procedure is shown in Algorithm 4. The learning algorithm is nearly identical, but we also maintain a vector of the sum of the weights, $\boldsymbol{m}$. At the end of the learning procedure, we divide this sum by the total number of updates t, to compute the average weights, $\overline{\boldsymbol{\theta}}$. These average weights are then used for prediction. In the algorithm sketch, the average is computed from a running sum, $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}$. However, this is inefficient, because it requires $|\boldsymbol{\theta}|$ operations to update the running sum. When $\boldsymbol{f}(\boldsymbol{x}, y)$ is sparse, $|\boldsymbol{\theta}| \gg |\boldsymbol{f}(\boldsymbol{x}, y)|$ for any individual $(\boldsymbol{x}, y)$. This means that computing the running sum will be much more expensive than computing of the update to $\boldsymbol{\theta}$ itself, which requires only $2 \times |\boldsymbol{f}(\boldsymbol{x}, y)|$ operations. One of the exercises is to sketch a more efficient algorithm for computing the averaged weights.

Even if the dataset is not separable, the averaged weights will eventually converge. One possible stopping criterion is to check the difference between the average weight vectors after each pass through the data: if the norm of the difference falls below some predefined threshold, we can stop training. Another stopping criterion is to hold out some data, and to measure the predictive accuracy on this heldout data. When the accuracy on the heldout data starts to decrease, the learning algorithm has begun to **overfit** the training set. At this point, it is probably best to stop; this stopping criterion is known as **early stopping**.

Generalization is the ability to make good predictions on instances that are not in the training data. Averaging can be proven to improve generalization, by computing an upper bound on the generalization error (Freund and Schapire, 1999; Collins, 2002).

2.4 Loss functions and large-margin classification

Naïve Bayes chooses the weights $\boldsymbol{\theta}$ by maximizing the joint log-likelihood $\log \mathrm{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)})$. By convention, optimization problems are generally formulated as minimization of a **loss function**. The input to a loss function is the vector of weights $\boldsymbol{\theta}$, and the output is a

Algorithm 4 Averaged perceptron learning algorithm

```
procedure AVG-PERCEPTRON(x^(1:N), y^(1:N))
    t ← 0
    θ^(0) ← 0
    repeat
        t ← t + 1
        Select an instance i
        ŷ ← argmax_y θ^(t−1) · f(x^(i), y)
        if ŷ ≠ y^(i) then
            θ^(t) ← θ^(t−1) + f(x^(i), y^(i)) − f(x^(i), ŷ)
        else
            θ^(t) ← θ^(t−1)
        m ← m + θ^(t)
    until tired
    θ̄ ← (1/t) m
    return θ̄
```

non-negative number, measuring the performance of the classifier on a training instance. Formally, the loss $\ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$ is then a measure of the performance of the weights $\boldsymbol{\theta}$ on the instance $(\boldsymbol{x}^{(i)}, y^{(i)})$. The goal of learning is to minimize the sum of the losses across all instances in the training set.

We can trivially reformulate maximum likelihood as a loss function, by defining the loss function to be the *negative* log-likelihood:

$$\log \mathrm{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{2.36}$$

$$\ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = -\log \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{2.37}$$

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmin}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.38}$$

$$= \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \tag{2.39}$$

The problem of minimizing ℓ_{NB} is thus identical to maximum-likelihood estimation.

Loss functions provide a general framework for comparing learning objectives. For example, an alternative loss function is the **zero-one loss**,

$$\ell_{0\text{-}1}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & y^{(i)} = \operatorname{argmax}_y \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \\ 1, & \text{otherwise} \end{cases} \tag{2.40}$$

The zero-one loss is zero if the instance is correctly classified, and one otherwise. The sum of zero-one losses is proportional to the error rate of the classifier on the training data. Since a low error rate is often the ultimate goal of classification, this may seem ideal. But the zero-one loss has several problems. One is that it is **non-convex**,[14] which means that there is no guarantee that gradient-based optimization will be effective. A more serious problem is that the derivatives are useless: the partial derivative with respect to any parameter is zero everywhere, except at the points where $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$ for some $\hat{y}$. At those points, the loss is discontinuous, and the derivative is undefined.

The perceptron optimizes a loss function that has better properties for learning:

$$\ell_{\text{PERCEPTRON}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \max_{y \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}), \quad [2.41]$$

When $\hat{y} = y^{(i)}$, the loss is zero; otherwise, it increases linearly with the gap between the score for the predicted label $\hat{y}$ and the score for the true label $y^{(i)}$. Plotting this loss against the input $\max_{y \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$ gives a hinge shape, motivating the name **hinge loss**.

To see why this is the loss function optimized by the perceptron, take the derivative with respect to $\boldsymbol{\theta}$,

$$\frac{\partial}{\partial \boldsymbol{\theta}} \ell_{\text{PERCEPTRON}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}). \quad [2.42]$$

At each instance, the perceptron algorithm takes a step of magnitude one in the opposite direction of this **gradient**, $\nabla_{\boldsymbol{\theta}} \ell_{\text{PERCEPTRON}} = \frac{\partial}{\partial \boldsymbol{\theta}} \ell_{\text{PERCEPTRON}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$. As we will see in § 2.6, this is an example of the optimization algorithm **stochastic gradient descent**, applied to the objective in Equation 2.41.

***Breaking ties with subgradient descent** [15] Careful readers will notice the tacit assumption that there is a unique $\hat{y}$ that maximizes $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$. What if there are two or more labels that maximize this function? Consider binary classification: if the maximizer is $y^{(i)}$, then the gradient is zero, and so is the perceptron update; if the maximizer is $\hat{y} \neq y^{(i)}$, then the update is the difference $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$. The underlying issue is that the perceptron loss is not **smooth**, because the first derivative has a discontinuity at the hinge point, where the score for the true label $y^{(i)}$ is equal to the score for some other label $\hat{y}$. At this point, there is no unique gradient; rather, there is a set of **subgradients**. A vector $\boldsymbol{v}$ is

[14] A function f is **convex** iff $\alpha f(x_i) + (1-\alpha) f(x_j) \geq f(\alpha x_i + (1-\alpha) x_j)$, for all $\alpha \in [0, 1]$ and for all x_i and x_j on the domain of the function. In words, any weighted average of the output of f applied to any two points is larger than the output of f when applied to the weighted average of the same two points. Convexity implies that any local minimum is also a global minimum, and there are many effective techniques for optimizing convex functions (Boyd and Vandenberghe, 2004). See Appendix B for a brief review.

[15] Throughout this text, advanced topics will be marked with an asterisk.

a subgradient of the function g at $\boldsymbol{u}_0$ iff $g(\boldsymbol{u}) - g(\boldsymbol{u}_0) \geq \boldsymbol{v} \cdot (\boldsymbol{u} - \boldsymbol{u}_0)$ for all $\boldsymbol{u}$. Graphically, this defines the set of hyperplanes that include $g(\boldsymbol{u}_0)$ and do not intersect g at any other point. As we approach the hinge point from the left, the gradient is $\boldsymbol{f}(\boldsymbol{x}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}, y)$; as we approach from the right, the gradient is $\mathbf{0}$. At the hinge point, the subgradients include all vectors that are bounded by these two extremes. In subgradient descent, *any* subgradient can be used (Bertsekas, 2012). Since both $\mathbf{0}$ and $\boldsymbol{f}(\boldsymbol{x}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}, y)$ are subgradients at the hinge point, either one can be used in the perceptron update. This means that if multiple labels maximize $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$, any of them can be used in the perceptron update.

Perceptron versus Naïve Bayes The perceptron loss function has some pros and cons with respect to the negative log-likelihood loss implied by Naïve Bayes.

- Both ℓ_{NB} and $\ell_{\text{PERCEPTRON}}$ are convex, making them relatively easy to optimize. However, ℓ_{NB} can be optimized in closed form, while $\ell_{\text{PERCEPTRON}}$ requires iterating over the dataset multiple times.
- ℓ_{NB} can suffer *infinite* loss on a single example, since the logarithm of zero probability is negative infinity. Naïve Bayes will therefore overemphasize some examples, and underemphasize others.
- The Naïve Bayes classifier assumes that the observed features are conditionally independent, given the label, and the performance of the classifier depends on the extent to which this assumption holds. The perceptron requires no such assumption.
- $\ell_{\text{PERCEPTRON}}$ treats all correct answers equally. Even if $\boldsymbol{\theta}$ only gives the correct answer by a tiny margin, the loss is still zero.

2.4.1 Online large margin classification

This last comment suggests a potential problem with the perceptron. Suppose a test example is very close to a training example, but not identical. If the classifier only gets the correct answer on the training example by a small amount, then it may give a different answer on the nearby test instance. To formalize this intuition, define the **margin** as,

$$\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \max_{y \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y). \quad [2.43]$$

The margin represents the difference between the score for the correct label $y^{(i)}$, and the score for the highest-scoring incorrect label. The intuition behind **large margin classification** is that it is not enough to label the training data correctly — the correct label should be separated from other labels by a comfortable margin. This idea can be encoded

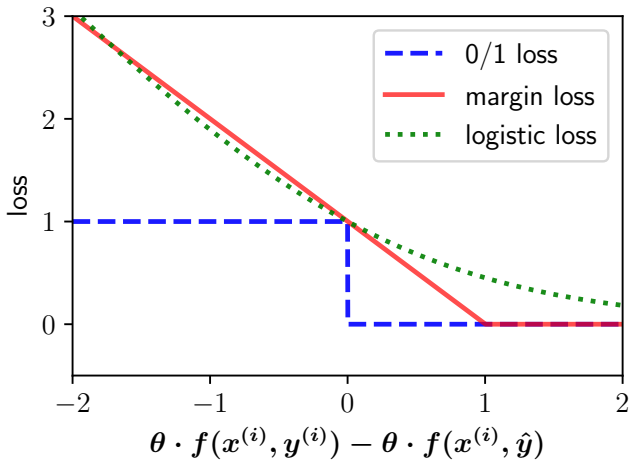


Figure 2.2: Margin, zero-one, and logistic loss functions.

into a loss function,

$$\ell_{\text{MARGIN}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \\ 1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), & \text{otherwise} \end{cases} \quad [2.44]$$

$$= \Big(1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})\Big)_+, \quad [2.45]$$

where $(x)_+ = \max(0, x)$. The loss is zero if there is a margin of at least 1 between the score for the true label and the best-scoring alternative $\hat{y}$. This is almost identical to the perceptron loss, but the hinge point is shifted to the right, as shown in Figure 2.2. The margin loss is a convex upper bound on the zero-one loss.

The margin loss can be minimized using an online learning rule that is similar to perceptron. We will call this learning rule the **online support vector machine**, for reasons that will be discussed in the derivation. Let us first generalize the notion of a classification error with a **cost function** $c(y^{(i)}, y)$. We will focus on the simple cost function,

$$c(y^{(i)}, y) = \begin{cases} 1, & y^{(i)} \neq \hat{y} \\ 0, & \text{otherwise}, \end{cases} \quad [2.46]$$

but it is possible to design specialized cost functions that assign heavier penalties to especially undesirable errors (Tsochantaridis et al., 2004). This idea is revisited in chapter 7.

Using the cost function, we can now define the online support vector machine as the

following classification rule:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y) \quad [2.47]$$

$$\boldsymbol{\theta}^{(t)} \leftarrow (1-\lambda)\boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) \quad [2.48]$$

This update is similar in form to the perceptron, with two key differences.

- Rather than selecting the label $\hat{y}$ that maximizes the score of the current classification model, the argmax searches for labels that are both *strong*, as measured by $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$, and *wrong*, as measured by $c(y^{(i)}, y)$. This maximization is known as **cost-augmented decoding**, because it augments the maximization objective to favor high-cost labels. If the highest-scoring label is $y = y^{(i)}$, then the margin loss for this instance is zero, and no update is needed. If not, then an update is required to reduce the margin loss — even if the current model classifies the instance correctly. Cost augmentation is only done while learning; it is not applied when making predictions on unseen data.
- The previous weights $\boldsymbol{\theta}^{(t-1)}$ are scaled by $(1-\lambda)$, with $\lambda \in (0, 1)$. The effect of this term is to cause the weights to "decay" back towards zero. In the support vector machine, this term arises from the minimization of a specific form of the margin, as described below. However, it can also be viewed as a form of **regularization**, which can help to prevent overfitting (see § 2.5.1). In this sense, it plays a role that is similar to smoothing in Naïve Bayes (see § 2.2.4).

2.4.2 *Derivation of the online support vector machine

The derivation of the online support vector machine is somewhat involved, but gives further intuition about why the method works. Begin by returning the idea of linear separability (Definition 1): if a dataset is linearly separable, then there is some hyperplane $\boldsymbol{\theta}$ that correctly classifies all training instances with margin ρ. This margin can be increased to any desired value by multiplying the weights by a constant.

Now, for any datapoint $(\boldsymbol{x}^{(i)}, y^{(i)})$, the geometric distance to the separating hyperplane is given by $\frac{\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{\theta}||_2}$, where the denominator is the norm of the weights, $||\boldsymbol{\theta}||_2 = \sqrt{\sum_j \theta_j^2}$. The geometric distance is sometimes called the **geometric margin**, in contrast to the **functional margin** $\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$. Both are shown in Figure 2.3. The geometric margin is a good measure of the robustness of the separator: if the functional margin is large, but the norm $||\boldsymbol{\theta}||_2$ is also large, then a small change in $\boldsymbol{x}^{(i)}$ could cause it to be misclassified. We therefore seek to maximize the minimum geometric margin across the dataset, subject

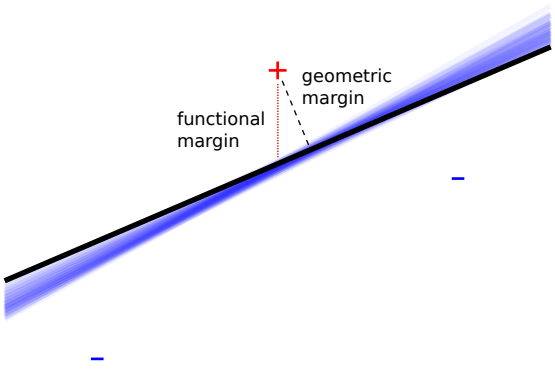


Figure 2.3: Functional and geometric margins for a binary classification problem. All separators that satisfy the margin constraint are shown. The separator with the largest geometric margin is shown in bold.

to the constraint that the margin loss is always zero:

$$\begin{aligned} \max_{\boldsymbol{\theta}} \quad & \min_{i=1,2,\ldots N} \quad \frac{\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{\theta}||_2} \\ \text{s.t.} \quad & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \quad \forall i. \end{aligned} \tag{2.49}$$

This is a **constrained optimization** problem, where the second line describes constraints on the space of possible solutions $\boldsymbol{\theta}$. In this case, the constraint is that the functional margin always be at least one, and the objective is that the minimum geometric margin be as large as possible.

Constrained optimization is reviewed in Appendix B. In this case, further manipulation yields an unconstrained optimization problem. First, note that the norm $||\boldsymbol{\theta}||_2$ scales linearly: $||a\boldsymbol{\theta}||_2 = a||\boldsymbol{\theta}||_2$. Furthermore, the functional margin γ is a linear function of $\boldsymbol{\theta}$, so that $\gamma(a\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) = a\gamma(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)})$. As a result, any scaling factor on $\boldsymbol{\theta}$ will cancel in the numerator and denominator of the geometric margin. If the data is linearly separable at any $\rho > 0$, it is always possible to rescale the functional margin to 1 by multiplying $\boldsymbol{\theta}$ by a scalar constant. We therefore need only minimize the denominator $||\boldsymbol{\theta}||_2$, subject to the constraint on the functional margin. The minimizer of $||\boldsymbol{\theta}||_2$ is also the minimizer of $\frac{1}{2}||\boldsymbol{\theta}||_2^2 = \frac{1}{2}\sum \theta_j^2$, which is easier to work with. This yields a simpler optimization prob-

lem:

$$\min_{\boldsymbol{\theta}} . \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2$$
$$\text{s.t.} \quad \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \quad \forall_i. \tag{2.50}$$

This problem is a **quadratic program**: the objective is a quadratic function of the parameters, and the constraints are all linear inequalities. One solution to this problem is to incorporate the constraints through Lagrange multipliers $\alpha_i \geq 0, i = 1, 2, \ldots, N$. The instances for which $\alpha_i > 0$ are called **support vectors**; other instances are irrelevant to the classification boundary. This motivates the name **support vector machine**.

Thus far we have assumed linear separability, but many datasets of interest are not linearly separable. In this case, there is no $\boldsymbol{\theta}$ that satisfies the margin constraint. To add more flexibility, we can introduce a set of **slack variables** $\xi_i \geq 0$. Instead of requiring that the functional margin be greater than or equal to one, we require that it be greater than or equal to $1 - \xi_i$. Ideally there would not be any slack, so the slack variables are penalized in the objective function:

$$\min_{\boldsymbol{\theta}, \boldsymbol{\xi}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 + C\sum_{i=1}^{N} \xi_i$$
$$\text{s.t.} \quad \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) + \xi_i \geq 1, \quad \forall i$$
$$\xi_i \geq 0, \quad \forall_i. \tag{2.51}$$

The hyperparameter C controls the tradeoff between violations of the margin constraint and the preference for a low norm of $\boldsymbol{\theta}$. As $C \to \infty$, slack is infinitely expensive, and there is only a solution if the data is separable. As $C \to 0$, slack becomes free, and there is a trivial solution at $\boldsymbol{\theta} = \mathbf{0}$. Thus, C plays a similar role to the smoothing parameter in Naïve Bayes (§ 2.2.4), trading off between a close fit to the training data and better generalization. Like the smoothing parameter of Naïve Bayes, C must be set by the user, typically by maximizing performance on a heldout development set.

To solve the constrained optimization problem defined in Equation 2.51, we can first solve for the slack variables,

$$\xi_i \geq (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+. \tag{2.52}$$

The inequality is tight: the optimal solution is to make the slack variables as small as possible, while still satisfying the constraints (Ratliff et al., 2007; Smith, 2011). By plugging in the minimum slack variables back into Equation 2.51, the problem can be transformed into the unconstrained optimization,

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_{i=1}^{N}(1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+, \tag{2.53}$$

where each ξ_i has been substituted by the right-hand side of Equation 2.52, and the factor of C on the slack variables has been replaced by an equivalent factor of $\lambda = \frac{1}{C}$ on the norm of the weights.

Equation 2.53 can be rewritten by expanding the margin,

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_{i=1}^{N} \left(\max_{y \in \mathcal{Y}} \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y) \right) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \right)_+, \tag{2.54}$$

where $c(y, y^{(i)})$ is the cost function defined in Equation 2.46. We can now differentiate with respect to the weights,

$$\nabla_{\boldsymbol{\theta}} L_{\text{SVM}} = \lambda \boldsymbol{\theta} + \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.55}$$

where L_{SVM} refers to minimization objective in Equation 2.54 and $\hat{y} = \text{argmax}_{y \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y)$. The online support vector machine update arises from the application of **stochastic gradient descent** (described in § 2.6.2) to this gradient.

2.5 Logistic regression

Thus far, we have seen two broad classes of learning algorithms. Naïve Bayes is a probabilistic method, where learning is equivalent to estimating a joint probability distribution. The perceptron and support vector machine are discriminative, error-driven algorithms: the learning objective is closely related to the number of errors on the training data. Probabilistic and error-driven approaches each have advantages: probability makes it possible to quantify uncertainty about the predicted labels, but the probability model of Naïve Bayes makes unrealistic independence assumptions that limit the features that can be used.

Logistic regression combines advantages of discriminative and probabilistic classifiers. Unlike Naïve Bayes, which starts from the **joint probability** $\text{p}_{X,Y}$, logistic regression defines the desired **conditional probability** $\text{p}_{Y|X}$ directly. Think of $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ as a scoring function for the compatibility of the base features $\boldsymbol{x}$ and the label y. To convert this score into a probability, we first exponentiate, obtaining $\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y))$, which is guaranteed to be non-negative. Next, we normalize, dividing over all possible labels $y' \in \mathcal{Y}$. The resulting conditional probability is defined as,

$$\text{p}(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y))}{\sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y'))}. \tag{2.56}$$

Given a dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, the weights $\boldsymbol{\theta}$ are estimated by **maximum conditional likelihood**,

$$\log \mathrm{p}(\boldsymbol{y}^{(1:N)} \mid \boldsymbol{x}^{(1:N)}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \quad [2.57]$$

$$= \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right). \quad [2.58]$$

The final line is obtained by plugging in Equation 2.56 and taking the logarithm.[16] Inside the sum, we have the (additive inverse of the) **logistic loss**,

$$\ell_{\text{LOGREG}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = -\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \log \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')) \quad [2.59]$$

The logistic loss is shown in Figure 2.2 on page 31. A key difference from the zero-one and hinge losses is that logistic loss is never zero. This means that the objective function can always be improved by assigning higher confidence to the correct label.

2.5.1 Regularization

As with the support vector machine, better generalization can be obtained by penalizing the norm of $\boldsymbol{\theta}$. This is done by adding a multiple of the squared norm $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$ to the minimization objective. This is called L_2 regularization, because $||\boldsymbol{\theta}||_2^2$ is the squared L_2 norm of the vector $\boldsymbol{\theta}$. Regularization forces the estimator to trade off performance on the training data against the norm of the weights, and this can help to prevent overfitting. Consider what would happen to the unregularized weight for a base feature j that is active in only one instance $\boldsymbol{x}^{(i)}$: the conditional log-likelihood could always be improved by increasing the weight for this feature, so that $\boldsymbol{\theta}_{(j,y^{(i)})} \rightarrow \infty$ and $\boldsymbol{\theta}_{(j,\tilde{y} \neq y^{(i)})} \rightarrow -\infty$, where (j, y) is the index of feature associated with $x_j^{(i)}$ and label y in $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$.

In § 2.2.4 (footnote 11), we saw that smoothing the probabilities of a Naïve Bayes classifier can be justified as a form of maximum a posteriori estimation, in which the parameters of the classifier are themselves random variables, drawn from a **prior distribution**. The same justification applies to L_2 regularization. In this case, the prior is a zero-mean Gaussian on each term of $\boldsymbol{\theta}$. The log-likelihood under a zero-mean Gaussian is,

$$\log N(\theta_j; 0, \sigma^2) \propto -\frac{1}{2\sigma^2}\theta_j^2, \quad [2.60]$$

so that the regularization weight λ is equal to the inverse variance of the prior, $\lambda = \frac{1}{\sigma^2}$.

[16] The log-sum-exp term is a common pattern in machine learning. It is numerically unstable, because it will underflow if the inner product is small, and overflow if the inner product is large. Scientific computing libraries usually contain special functions for computing `logsumexp`, but with some thought, you should be able to see how to create an implementation that is numerically stable.

2.5.2 Gradients

Logistic loss is minimized by optimization along the gradient. Specific algorithms are described in the next section, but first let's compute the gradient with respect to the logistic loss of a single example:

$$\ell_{\text{LOGREG}} = -\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \log \sum_{y' \in \mathcal{Y}} \exp\Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\Big) \tag{2.61}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \frac{1}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \sum_{y' \in \mathcal{Y}} \exp\Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\Big) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.62}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \frac{\exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.63}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.64}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + E_{Y|X}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]. \tag{2.65}$$

The final step employs the definition of a conditional expectation (§ A.5). The gradient of the logistic loss is equal to the difference between the expected counts under the current model, $E_{Y|X}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$, and the observed feature counts $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$. When these two vectors are equal for a single instance, there is nothing more to learn from it; when they are equal in sum over the entire dataset, there is nothing more to learn from the dataset as a whole. The gradient of the hinge loss is nearly identical, but it involves the features of the predicted label under the current model, $\boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$, rather than the expected features $E_{Y|X}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$ under the conditional distribution $\mathrm{p}(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

The regularizer contributes $\lambda\boldsymbol{\theta}$ to the overall gradient:

$$L_{\text{LOGREG}} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N} \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \tag{2.66}$$

$$\nabla_{\boldsymbol{\theta}} L_{\text{LOGREG}} = \lambda\boldsymbol{\theta} - \sum_{i=1}^{N} \Big(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]\Big). \tag{2.67}$$

2.6 Optimization

Each of the classification algorithms in this chapter can be viewed as an optimization problem:

- In Naïve Bayes, the objective is the joint likelihood $\log \mathrm{p}(\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)})$. Maximum likelihood estimation yields a closed-form solution for $\boldsymbol{\theta}$.

- In the support vector machine, the objective is the regularized margin loss,

$$L_{\text{SVM}} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_{i=1}^{N}(\max_{y\in\mathcal{Y}}(\boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y) + c(y^{(i)},y)) - \boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y^{(i)}))_+, \quad [2.68]$$

  There is no closed-form solution, but the objective is convex. The perceptron algorithm minimizes a similar objective.

- In logistic regression, the objective is the regularized negative log-likelihood,

$$L_{\text{LOGREG}} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N}\left(\boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y^{(i)}) - \log\sum_{y\in\mathcal{Y}}\exp\left(\boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y)\right)\right) \quad [2.69]$$

  Again, there is no closed-form solution, but the objective is convex.

These learning algorithms are distinguished by *what* is being optimized, rather than *how* the optimal weights are found. This decomposition is an essential feature of contemporary machine learning. The domain expert's job is to design an objective function — or more generally, a **model** of the problem. If the model has certain characteristics, then generic optimization algorithms can be used to find the solution. In particular, if an objective function is differentiable, then gradient-based optimization can be employed; if it is also convex, then gradient-based optimization is guaranteed to find the globally optimal solution. The support vector machine and logistic regression have both of these properties, and so are amenable to generic **convex optimization** techniques (Boyd and Vandenberghe, 2004).

2.6.1 Batch optimization

In **batch optimization**, each update to the weights is based on a computation involving the entire dataset. One such algorithm is **gradient descent**, which iteratively updates the weights,

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta^{(t)}\nabla_{\boldsymbol{\theta}}L, \quad [2.70]$$

where $\nabla_{\boldsymbol{\theta}}L$ is the gradient computed over the entire training set, and $\eta^{(t)}$ is the **learning rate** at iteration t. If the objective L is a convex function of $\boldsymbol{\theta}$, then this procedure is guaranteed to terminate at the global optimum, for appropriate schedule of learning rates, $\eta^{(t)}$.[17]

[17] Convergence proofs typically require the learning rate to satisfy the following conditions: $\sum_{t=1}^{\infty}\eta^{(t)} = \infty$ and $\sum_{t=1}^{\infty}(\eta^{(t)})^2 < \infty$ (Bottou et al., 2016). These properties are satisfied by any learning rate schedule $\eta^{(t)} = \eta^{(0)}t^{-\alpha}$ for $\alpha \in [1, 2]$.

In practice, gradient descent can be slow to converge, as the gradient can become infinitesimally small. Faster convergence can be obtained by second-order Newton optimization, which incorporates the inverse of the **Hessian matrix**,

$$H_{i,j} = \frac{\partial^2 L}{\partial \theta_i \partial \theta_j} \quad [2.71]$$

The size of the Hessian matrix is quadratic in the number of features. In the bag-of-words representation, this is usually too big to store, let alone invert. **Quasi-Network optimization** techniques maintain a low-rank approximation to the inverse of the Hessian matrix. Such techniques usually converge more quickly than gradient descent, while remaining computationally tractable even for large feature sets. A popular quasi-Newton algorithm is L-BFGS (Liu and Nocedal, 1989), which is implemented in many scientific computing environments, such as SCIPY and MATLAB.

For any gradient-based technique, the user must set the learning rates $\eta^{(t)}$. While convergence proofs usually employ a decreasing learning rate, in practice, it is common to fix $\eta^{(t)}$ to a small constant, like 10^{-3}. The specific constant can be chosen by experimentation, although there is research on determining the learning rate automatically (Schaul et al., 2013; Wu et al., 2018).

2.6.2 Online optimization

Batch optimization computes the objective on the entire training set before making an update. This may be inefficient, because at early stages of training, a small number of training examples could point the learner in the correct direction. **Online learning** algorithms make updates to the weights while iterating through the training data. The theoretical basis for this approach is a stochastic approximation to the true objective function,

$$\sum_{i=1}^{N} \ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \approx N \times \ell(\boldsymbol{\theta}; \boldsymbol{x}^{(j)}, y^{(j)}), \qquad (\boldsymbol{x}^{(j)}, y^{(j)}) \sim \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \quad [2.72]$$

where the instance $(\boldsymbol{x}^{(j)}, y^{(j)})$ is sampled at random from the full dataset.

In **stochastic gradient descent**, the approximate gradient is computed by randomly sampling a single instance, and an update is made immediately. This is similar to the perceptron algorithm, which also updates the weights one instance at a time. In **minibatch** stochastic gradient descent, the gradient is computed over a small set of instances. A typical approach is to set the minibatch size so that the entire batch fits in memory on a graphics processing unit (GPU; Neubig et al., 2017). It is then possible to speed up learning by parallelizing the computation of the gradient over each instance in the minibatch.

Algorithm 5 offers a generalized view of gradient descent. In standard gradient descent, the batcher returns a single batch with all the instances. In stochastic gradient de-

Algorithm 5 Generalized gradient descent. The function BATCHER partitions the training set into B batches such that each instance appears in exactly one batch. In gradient descent, $B = 1$; in stochastic gradient descent, $B = N$; in minibatch stochastic gradient descent, $1 < B < N$.

procedure GRADIENT-DESCENT($\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}, L, \eta^{(1...\infty)}$, BATCHER, T_{max})
  $\boldsymbol{\theta} \leftarrow \mathbf{0}$
  $t \leftarrow 0$
  repeat
    $(\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(B)}) \leftarrow$ BATCHER(N)
    for $n \in \{1, 2, \ldots, B\}$ **do**
      $t \leftarrow t + 1$
      $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \eta^{(t)} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{(t-1)}; \boldsymbol{x}^{(b_1^{(n)}, b_2^{(n)}, \ldots)}, \boldsymbol{y}^{(b_1^{(n)}, b_2^{(n)}, \ldots)})$
      if Converged$(\boldsymbol{\theta}^{(1,2,\ldots,t)})$ **then**
        return $\boldsymbol{\theta}^{(t)}$
  until $t \geq T_{\text{max}}$
  return $\boldsymbol{\theta}^{(t)}$

scent, it returns N batches with one instance each. In mini-batch settings, the batcher returns B minibatches, $1 < B < N$.

There are many other techniques for online learning, and research in this area is ongoing (Bottou et al., 2016). Some algorithms use an adaptive learning rate, which can be different for every feature (Duchi et al., 2011). Features that occur frequently are likely to be updated frequently, so it is best to use a small learning rate; rare features will be updated infrequently, so it is better to take larger steps. The **AdaGrad** (adaptive gradient) algorithm achieves this behavior by storing the sum of the squares of the gradients for each feature, and rescaling the learning rate by its inverse:

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{(t)}; \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.73}$$

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \frac{\eta^{(t)}}{\sqrt{\sum_{t'=1}^{t} g_{t,j}^2}} g_{t,j}, \tag{2.74}$$

where j iterates over features in $\boldsymbol{f}(\boldsymbol{x}, y)$.

In most cases, the number of active features for any instance is much smaller than the number of weights. If so, the computation cost of online optimization will be dominated by the update from the regularization term, $\lambda\boldsymbol{\theta}$. The solution is to be "lazy", updating each θ_j only as it is used. To implement lazy updating, store an additional parameter τ_j, which is the iteration at which θ_j was last updated. If θ_j is needed at time t, the $t - \tau$ regularization updates can be performed all at once. This strategy is described in detail by Kummerfeld et al. (2015).

2.7 *Additional topics in classification

This section presents some additional topics in classification that are particularly relevant for natural language processing, especially for understanding the research literature.

2.7.1 Feature selection by regularization

In logistic regression and large-margin classification, generalization can be improved by regularizing the weights towards 0, using the L_2 norm. But rather than encouraging weights to be small, it might be better for the model to be **sparse**: it should assign weights of exactly zero to most features, and only assign non-zero weights to features that are clearly necessary. This idea can be formalized by the L_0 norm, $L_0 = ||\boldsymbol{\theta}||_0 = \sum_j \delta(\theta_j \neq 0)$, which applies a constant penalty for each non-zero weight. This norm can be thought of as a form of **feature selection**: optimizing the L_0-regularized conditional likelihood is equivalent to trading off the log-likelihood against the number of active features. Reducing the number of active features is desirable because the resulting model will be fast, low-memory, and should generalize well, since irrelevant features will be pruned away. Unfortunately, the L_0 norm is non-convex and non-differentiable. Optimization under L_0 regularization is **NP-hard**, meaning that it can be solved efficiently only if P=NP (Ge et al., 2011).

A useful alternative is the L_1 norm, which is equal to the sum of the absolute values of the weights, $||\boldsymbol{\theta}||_1 = \sum_j |\theta_j|$. The L_1 norm is convex, and can be used as an approximation to L_0 (Tibshirani, 1996). Conveniently, the L_1 norm also performs feature selection, by driving many of the coefficients to zero; it is therefore known as a **sparsity inducing regularizer**. The L_1 norm does not have a gradient at $\theta_j = 0$, so we must instead optimize the L_1-regularized objective using **subgradient** methods. The associated stochastic subgradient descent algorithms are only somewhat more complex than conventional SGD; Sra et al. (2012) survey approaches for estimation under L_1 and other regularizers.

Gao et al. (2007) compare L_1 and L_2 regularization on a suite of NLP problems, finding that L_1 regularization generally gives similar accuracy to L_2 regularization, but that L_1 regularization produces models that are between ten and fifty times smaller, because more than 90% of the feature weights are set to zero.

2.7.2 Other views of logistic regression

In binary classification, we can dispense with the feature function, and choose y based on the inner product of $\boldsymbol{\theta} \cdot \boldsymbol{x}$. The conditional probability $\mathrm{p}_{Y|X}$ is obtained by passing this

inner product through a **logistic function**,

$$\sigma(a) \triangleq \frac{\exp(a)}{1+\exp(a)} = (1+\exp(-a))^{-1} \quad [2.75]$$

$$\mathrm{p}(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta} \cdot \boldsymbol{x}). \quad [2.76]$$

This is the origin of the name "logistic regression." Logistic regression can be viewed as part of a larger family of **generalized linear models** (GLMs), in which various other **link functions** convert between the inner product $\boldsymbol{\theta} \cdot \boldsymbol{x}$ and the parameter of a conditional probability distribution.

Logistic regression and related models are sometimes referred to as **log-linear**, because the log-probability is a linear function of the features. But in the early NLP literature, logistic regression was often called **maximum entropy** classification (Berger et al., 1996). This name refers to an alternative formulation, in which the goal is to find the maximum entropy probability function that satisfies **moment-matching** constraints. These constraints specify that the empirical counts of each feature should match the expected counts under the induced probability distribution $\mathrm{p}_{Y|X;\boldsymbol{\theta}}$,

$$\sum_{i=1}^{N} f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}(y \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{x}^{(i)}, y), \quad \forall j \quad [2.77]$$

The moment-matching constraint is satisfied exactly when the derivative of the conditional log-likelihood function (Equation 2.65) is equal to zero. However, the constraint can be met by many values of $\boldsymbol{\theta}$, so which should we choose?

The **entropy** of the conditional probability distribution $\mathrm{p}_{Y|X}$ is,

$$H(\mathrm{p}_{Y|X}) = -\sum_{\boldsymbol{x} \in \mathcal{X}} \mathrm{p}_X(\boldsymbol{x}) \sum_{y \in \mathcal{Y}} \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}) \log \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}), \quad [2.78]$$

where $\mathcal{X}$ is the set of all possible feature vectors, and $\mathrm{p}_X(\boldsymbol{x})$ is the probability of observing the base features $\boldsymbol{x}$. The distribution p_X is unknown, but it can be estimated by summing over all the instances in the training set,

$$\tilde{H}(\mathrm{p}_{Y|X}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}^{(i)}) \log \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}^{(i)}). \quad [2.79]$$

If the entropy is large, the likelihood function is smooth across possible values of y; if it is small, the likelihood function is sharply peaked at some preferred value; in the limiting case, the entropy is zero if $\mathrm{p}(y \mid x) = 1$ for some y. The maximum-entropy criterion chooses to make the weakest commitments possible, while satisfying the moment-matching constraints from Equation 2.77. The solution to this constrained optimization problem is identical to the maximum conditional likelihood (logistic-loss) formulation that was presented in § 2.5.

2.8 Summary of learning algorithms

It is natural to ask which learning algorithm is best, but the answer depends on what characteristics are important to the problem you are trying to solve.

Naïve Bayes *Pros*: easy to implement; estimation is fast, requiring only a single pass over the data; assigns probabilities to predicted labels; controls overfitting with smoothing parameter. *Cons*: often has poor accuracy, especially with correlated features.

Perceptron *Pros*: easy to implement; online; error-driven learning means that accuracy is typically high, especially after averaging. *Cons*: not probabilistic; hard to know when to stop learning; lack of margin can lead to overfitting.

Support vector machine *Pros*: optimizes an error-based metric, usually resulting in high accuracy; overfitting is controlled by a regularization parameter. *Cons*: not probabilistic.

Logistic regression *Pros*: error-driven and probabilistic; overfitting is controlled by a regularization parameter. *Cons*: batch learning requires black-box optimization; logistic loss can "overtrain" on correctly labeled examples.

One of the main distinctions is whether the learning algorithm offers a probability over labels. This is useful in modular architectures, where the output of one classifier is the input for some other system. In cases where probability is not necessary, the support vector machine is usually the right choice, since it is no more difficult to implement than the perceptron, and is often more accurate. When probability is necessary, logistic regression is usually more accurate than Naïve Bayes.

Additional resources

A machine learning textbook will offer more classifiers and more details (e.g., Murphy, 2012), although the notation will differ slightly from what is typical in natural language processing. Probabilistic methods are surveyed by Hastie et al. (2009), and Mohri et al. (2012) emphasize theoretical considerations. Bottou et al. (2016) surveys the rapidly moving field of online learning, and Kummerfeld et al. (2015) empirically review several optimization algorithms for large-margin learning. The python toolkit SCIKIT-LEARN includes implementations of all of the algorithms described in this chapter (Pedregosa et al., 2011).

Appendix B describes an alternative large-margin classifier, called **passive-aggressive**. Passive-aggressive is an online learner that seeks to make the smallest update that satisfies the margin constraint at the current instance. It is closely related to MIRA, which was used widely in NLP in the 2000s (Crammer and Singer, 2003).

Exercises

There will be exercises at the end of each chapter. In this chapter, the exercises are mostly mathematical, matching the subject material. In other chapters, the exercises will emphasize linguistics or programming.

1. Let $\boldsymbol{x}$ be a bag-of-words vector such that $\sum_{j=1}^{V} x_j = 1$. Verify that the multinomial probability $\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi})$, as defined in Equation 2.12, is identical to the probability of the same document under a categorical distribution, $\text{p}_{\text{cat}}(\boldsymbol{w}; \boldsymbol{\phi})$.

2. Suppose you have a single feature x, with the following conditional distribution:

$$\text{p}(x \mid y) = \begin{cases} \alpha, & X = 0, Y = 0 \\ 1 - \alpha, & X = 1, Y = 0 \\ 1 - \beta, & X = 0, Y = 1 \\ \beta, & X = 1, Y = 1. \end{cases} \quad [2.80]$$

   Further suppose that the prior is uniform, $\Pr(Y = 0) = \Pr(Y = 1) = \frac{1}{2}$, and that both $\alpha > \frac{1}{2}$ and $\beta > \frac{1}{2}$. Given a Naïve Bayes classifier with accurate parameters, what is the probability of making an error?

3. Derive the maximum-likelihood estimate for the parameter $\boldsymbol{\mu}$ in Naïve Bayes.

4. The classification models in the text have a vector of weights for each possible label. While this is notationally convenient, it is overdetermined: for any linear classifier that can be obtained with $K \times V$ weights, an equivalent classifier can be constructed using $(K - 1) \times V$ weights.

   a) Describe how to construct this classifier. Specifically, if given a set of weights $\boldsymbol{\theta}$ and a feature function $\boldsymbol{f}(\boldsymbol{x}, y)$, explain how to construct alternative weights and feature function $\boldsymbol{\theta}'$ and $\boldsymbol{f}'(\boldsymbol{x}, y)$, such that,

$$\forall y, y' \in \mathcal{Y}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y') = \boldsymbol{\theta}' \cdot \boldsymbol{f}'(\boldsymbol{x}, y) - \boldsymbol{\theta}' \cdot \boldsymbol{f}'(\boldsymbol{x}, y'). \quad [2.81]$$

   b) Explain how your construction justifies the well-known alternative form for binary logistic regression, $\Pr(Y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \frac{1}{1+\exp(-\boldsymbol{\theta}' \cdot \boldsymbol{x})} = \sigma(\boldsymbol{\theta}' \cdot \boldsymbol{x})$, where σ is the sigmoid function.

5. Suppose you have two labeled datasets D_1 and D_2, with the same features and labels.

   - Let $\boldsymbol{\theta}^{(1)}$ be the unregularized logistic regression (LR) coefficients from training on dataset D_1.

- Let $\boldsymbol{\theta}^{(2)}$ be the unregularized LR coefficients (same model) from training on dataset D_2.
- Let $\boldsymbol{\theta}^*$ be the unregularized LR coefficients from training on the combined dataset $D_1 \cup D_2$.

Under these conditions, prove that for any feature j,

$$\theta_j^* \geq \min(\theta_j^{(1)}, \theta_j^{(2)})$$
$$\theta_j^* \leq \max(\theta_j^{(1)}, \theta_j^{(2)}).$$

6. Let $\hat{\boldsymbol{\theta}}$ be the solution to an unregularized logistic regression problem, and let $\boldsymbol{\theta}^*$ be the solution to the same problem, with L_2 regularization. Prove that $||\boldsymbol{\theta}^*||_2^2 \leq ||\hat{\boldsymbol{\theta}}||_2^2$.

7. As noted in the discussion of averaged perceptron in § 2.3.2, the computation of the running sum $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}$ is unnecessarily expensive, requiring $K \times V$ operations. Give an alternative way to compute the averaged weights $\overline{\boldsymbol{\theta}}$, with complexity that is independent of V and linear in the sum of feature sizes $\sum_{i=1}^{N} |\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})|$.

8. Consider a dataset that is comprised of two identical instances $\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(2)}$ with distinct labels $y^{(1)} \neq y^{(2)}$. Assume all features are binary, $x_j \in \{0, 1\}$ for all j.

   Now suppose that the averaged perceptron always trains on the instance $(\boldsymbol{x}^{i(t)}, y^{i(t)})$, where $i(t) = 2 - (t \mod 2)$, which is 1 when the training iteration t is odd, and 2 when t is even. Further suppose that learning terminates under the following condition:

   $$\epsilon \geq \max_j \left| \frac{1}{t} \sum_t \theta_j^{(t)} - \frac{1}{t-1} \sum_t \theta_j^{(t-1)} \right|. \quad [2.82]$$

   In words, the algorithm stops when the largest change in the averaged weights is less than or equal to ϵ. Compute the number of iterations before the averaged perceptron terminates.

9. Prove that the margin loss is convex in $\boldsymbol{\theta}$. Use this definition of the margin loss:

   $$L(\boldsymbol{\theta}) = -\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y^*) + \max_y \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) + c(y^*, y), \quad [2.83]$$

   where y^* is the gold label. As a reminder, a function f is convex iff,

   $$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha) f(x_2), \quad [2.84]$$

   for any x_1, x_2 and $\alpha \in [0, 1]$.

10. If a function f is m-strongly convex, then for some $m > 0$, the following inequality holds for all x and x' on the domain of the function:

$$f(x') \leq f(x) + (\nabla_x f) \cdot (x' - x) + \frac{m}{2}||x' - x||_2^2. \quad [2.85]$$

Let $f(x) = L(\boldsymbol{\theta}^{(t)})$, representing the loss of the classifier at iteration t of gradient descent; let $f(x') = L(\boldsymbol{\theta}^{(t+1)})$. Assuming the loss function is m-convex, prove that $L(\boldsymbol{\theta}^{(t+1)}) \leq L(\boldsymbol{\theta}^{(t)})$ for an appropriate constant learning rate η, which will depend on m. Explain why this implies that gradient descent converges when applied to an m-strongly convex loss function with a unique minimum.

Chapter 3

Nonlinear classification

Linear classification may seem like all we need for natural language processing. The bag-of-words representation is inherently high dimensional, and the number of features is often larger than the number of labeled training instances. This means that it is usually possible to find a linear classifier that perfectly fits the training data, or even to fit any arbitrary labeling of the training instances! Moving to nonlinear classification may therefore only increase the risk of overfitting. Furthermore, for many tasks, **lexical features** (words) are meaningful in isolation, and can offer independent evidence about the instance label — unlike computer vision, where individual pixels are rarely informative, and must be evaluated holistically to make sense of an image. For these reasons, natural language processing has historically focused on linear classification.

But in recent years, nonlinear classifiers have swept through natural language processing, and are now the default approach for many tasks (Manning, 2015). There are at least three reasons for this change.

- There have been rapid advances in **deep learning**, a family of nonlinear methods that learn complex functions of the input through multiple layers of computation (Goodfellow et al., 2016).
- Deep learning facilitates the incorporation of **word embeddings**, which are dense vector representations of words. Word embeddings can be learned from large amounts of unlabeled data, and enable generalization to words that do not appear in the annotated training data (word embeddings are discussed in detail in chapter 14).
- While CPU speeds have plateaued, there have been rapid advances in specialized hardware called graphics processing units (GPUs), which have become faster, cheaper, and easier to program. Many deep learning models can be implemented efficiently on GPUs, offering substantial performance improvements over CPU-based computing.

This chapter focuses on **neural networks**, which are the dominant approach for nonlinear classification in natural language processing today.[1] Historically, a few other nonlinear learning methods have been applied to language data.

- **Kernel methods** are generalizations of the **nearest-neighbor** classification rule, which classifies each instance by the label of the most similar example in the training set. The application of the **kernel support vector machine** to information extraction is described in chapter 17.
- **Decision trees** classify instances by checking a set of conditions. Scaling decision trees to bag-of-words inputs is difficult, but decision trees have been successful in problems such as coreference resolution (chapter 15), where more compact feature sets can be constructed (Soon et al., 2001).
- **Boosting** and related **ensemble methods** work by combining the predictions of several "weak" classifiers, each of which may consider only a small subset of features. Boosting has been successfully applied to text classification (Schapire and Singer, 2000) and syntactic analysis (Abney et al., 1999), and remains one of the most successful methods on machine learning competition sites such as Kaggle (Chen and Guestrin, 2016).

Hastie et al. (2009) provide an excellent overview of these techniques.

3.1 Feedforward neural networks

Consider the problem of building a classifier for movie reviews. The goal is to predict a label $y \in \{\text{GOOD}, \text{BAD}, \text{OKAY}\}$ from a representation of the text of each document, $\boldsymbol{x}$. But what makes a good movie? The story, acting, cinematography, editing, soundtrack, and so on. Now suppose the training set contains labels for each of these additional features, $\boldsymbol{z} = [z_1, z_2, \ldots, z_{K_z}]^\top$. With a training set of such information, we could build a two-step classifier:

1. **Use the text $\boldsymbol{x}$ to predict the features $\boldsymbol{z}$.** Specifically, train a logistic regression classifier to compute $\text{p}(z_k \mid \boldsymbol{x})$, for each $k \in \{1, 2, \ldots, K_z\}$.
2. **Use the features $\boldsymbol{z}$ to predict the label y.** Again, train a logistic regression classifier to compute $\text{p}(y \mid \boldsymbol{z})$. On test data, $\boldsymbol{z}$ is unknown, so we will use the probabilities $\text{p}(\boldsymbol{z} \mid \boldsymbol{x})$ from the first layer as the features.

This setup is shown in Figure 3.1, which describes the proposed classifier in a **computation graph**: the text features $\boldsymbol{x}$ are connected to the middle layer $\boldsymbol{z}$, which is connected to the label y.

[1] I will use "deep learning" and "neural networks" interchangeably.

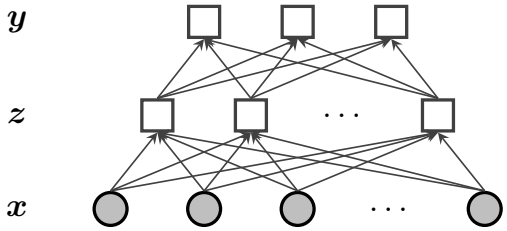


Figure 3.1: A feedforward neural network. Shaded circles indicate observed features, usually words; squares indicate nodes in the computation graph, which are computed from the information carried over the incoming arrows.

If we assume that each z_k is binary, $z_k \in \{0, 1\}$, then the probability $\text{p}(z_k \mid \boldsymbol{x})$ can be modeled using binary logistic regression:

$$\Pr(z_k = 1 \mid \boldsymbol{x}; \boldsymbol{\Theta}^{(x \rightarrow z)}) = \sigma(\boldsymbol{\theta}_k^{(x \rightarrow z)} \cdot \boldsymbol{x}) = (1 + \exp(-\boldsymbol{\theta}_k^{(x \rightarrow z)} \cdot \boldsymbol{x}))^{-1}, \quad [3.1]$$

where σ is the **sigmoid** function (shown in Figure 3.2), and the matrix $\boldsymbol{\Theta}^{(x \rightarrow z)} \in \mathbb{R}^{K_z \times V}$ is constructed by stacking the weight vectors for each z_k,

$$\boldsymbol{\Theta}^{(x \rightarrow z)} = [\boldsymbol{\theta}_1^{(x \rightarrow z)}, \boldsymbol{\theta}_2^{(x \rightarrow z)}, \ldots, \boldsymbol{\theta}_{K_z}^{(x \rightarrow z)}]^\top. \quad [3.2]$$

We will assume that $\boldsymbol{x}$ contains a term with a constant value of 1, so that a corresponding offset parameter is included in each $\boldsymbol{\theta}_k^{(x \rightarrow z)}$.

The output layer is computed by the multi-class logistic regression probability,

$$\Pr(y = j \mid \boldsymbol{z}; \boldsymbol{\Theta}^{(z \rightarrow y)}, \boldsymbol{b}) = \frac{\exp(\boldsymbol{\theta}_j^{(z \rightarrow y)} \cdot \boldsymbol{z} + b_j)}{\sum_{j' \in \mathcal{Y}} \exp(\boldsymbol{\theta}_{j'}^{(z \rightarrow y)} \cdot \boldsymbol{z} + b_{j'})}, \quad [3.3]$$

where b_j is an offset for label j, and the output weight matrix $\boldsymbol{\Theta}^{(z \rightarrow y)} \in \mathbb{R}^{K_y \times K_z}$ is again constructed by concatenation,

$$\boldsymbol{\Theta}^{(z \rightarrow y)} = [\boldsymbol{\theta}_1^{(z \rightarrow y)}, \boldsymbol{\theta}_2^{(z \rightarrow y)}, \ldots, \boldsymbol{\theta}_{K_y}^{(z \rightarrow y)}]^\top. \quad [3.4]$$

The vector of probabilities over each possible value of y is denoted,

$$\text{p}(\boldsymbol{y} \mid \boldsymbol{z}; \boldsymbol{\Theta}^{(z \rightarrow y)}, \boldsymbol{b}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z} + \boldsymbol{b}), \quad [3.5]$$

where element j in the output of the **SoftMax** function is computed as in Equation 3.3.

This set of equations defines a multilayer classifier, which can be summarized as,

$$\text{p}(\boldsymbol{z} \mid \boldsymbol{x}; \boldsymbol{\Theta}^{(x \rightarrow z)}) = \sigma(\boldsymbol{\Theta}^{(x \rightarrow z)} \boldsymbol{x}) \quad [3.6]$$

$$\text{p}(\boldsymbol{y} \mid \boldsymbol{z}; \boldsymbol{\Theta}^{(z \rightarrow y)}, \boldsymbol{b}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z} + \boldsymbol{b}), \quad [3.7]$$

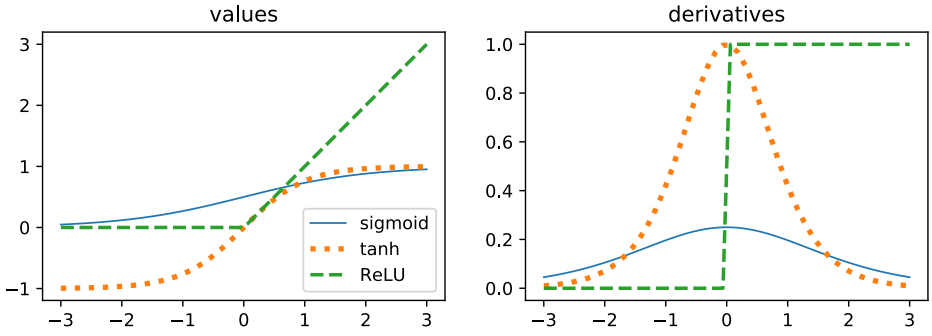


Figure 3.2: The sigmoid, tanh, and ReLU activation functions

where the function σ is now applied **elementwise** to the vector of inner products,

$$\sigma(\mathbf{\Theta}^{(x\rightarrow z)}\boldsymbol{x}) = [\sigma(\boldsymbol{\theta}_1^{(x\rightarrow z)} \cdot \boldsymbol{x}), \sigma(\boldsymbol{\theta}_2^{(x\rightarrow z)} \cdot \boldsymbol{x}), \ldots, \sigma(\boldsymbol{\theta}_{K_z}^{(x\rightarrow z)} \cdot \boldsymbol{x})]^\top. \quad [3.8]$$

Now suppose that the hidden features $\boldsymbol{z}$ are never observed, even in the training data. We can still construct the architecture in Figure 3.1. Instead of predicting y from a discrete vector of predicted values $\boldsymbol{z}$, we use the probabilities $\sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x})$. The resulting classifier is barely changed:

$$\boldsymbol{z} = \sigma(\mathbf{\Theta}^{(x\rightarrow z)}\boldsymbol{x}) \quad [3.9]$$

$$\text{p}(y \mid \boldsymbol{x}; \mathbf{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) = \text{SoftMax}(\mathbf{\Theta}^{(z\rightarrow y)}\boldsymbol{z} + \boldsymbol{b}). \quad [3.10]$$

This defines a classification model that predicts the label $y \in \mathcal{Y}$ from the base features $\boldsymbol{x}$, through a"hidden layer" $\boldsymbol{z}$. This is a **feedforward neural network**.[2]

3.2 Designing neural networks

There several ways to generalize the feedforward neural network.

3.2.1 Activation functions

If the hidden layer is viewed as a set of latent features, then the sigmoid function in Equation 3.9 represents the extent to which each of these features is "activated" by a given input. However, the hidden layer can be regarded more generally as a nonlinear transformation of the input. This opens the door to many other activation functions, some of which are shown in Figure 3.2. At the moment, the choice of activation functions is more art than science, but a few points can be made about the most popular varieties:

[2]The architecture is sometimes called a **multilayer perceptron**, but this is misleading, because each layer is not a perceptron as defined in the previous chapter.

- The range of the sigmoid function is $(0, 1)$. The bounded range ensures that a cascade of sigmoid functions will not "blow up" to a huge output, and this is important for deep networks with several hidden layers. The derivative of the sigmoid is $\frac{\partial}{\partial a}\sigma(a) = \sigma(a)(1 - \sigma(a))$. This derivative becomes small at the extremes, which can make learning slow; this is called the **vanishing gradient** problem.
- The range of the **tanh activation function** is $(-1, 1)$: like the sigmoid, the range is bounded, but unlike the sigmoid, it includes negative values. The derivative is $\frac{\partial}{\partial a}\tanh(a) = 1 - \tanh(a)^2$, which is steeper than the logistic function near the origin (LeCun et al., 1998). The tanh function can also suffer from vanishing gradients at extreme values.
- The **rectified linear unit (ReLU)** is zero for negative inputs, and linear for positive inputs (Glorot et al., 2011),

$$\text{ReLU}(a) = \begin{cases} a, & a \geq 0 \\ 0, & \text{otherwise.} \end{cases} \quad [3.11]$$

  The derivative is a step function, which is 1 if the input is positive, and zero otherwise. Once the activation is zero, the gradient is also zero. This can lead to the problem of "dead neurons", where some ReLU nodes are zero for all inputs, throughout learning. A solution is the **leaky ReLU**, which has a small positive slope for negative inputs (Maas et al., 2013),

$$\text{Leaky-ReLU}(a) = \begin{cases} a, & a \geq 0 \\ .0001a, & \text{otherwise.} \end{cases} \quad [3.12]$$

Sigmoid and tanh are sometimes described as **squashing functions**, because they squash an unbounded input into a bounded range. Glorot and Bengio (2010) recommend against the use of the sigmoid activation in deep networks, because its mean value of $\frac{1}{2}$ can cause the next layer of the network to be saturated, leading to small gradients on its own parameters. Several other activation functions are reviewed in the textbook by Goodfellow et al. (2016), who recommend ReLU as the "default option."

3.2.2 Network structure

Deep networks stack up several hidden layers, with each $\boldsymbol{z}^{(d)}$ acting as the input to the next layer, $\boldsymbol{z}^{(d+1)}$. As the total number of nodes in the network increases, so does its capacity to learn complex functions of the input. Given a fixed number of nodes, one must decide whether to emphasize width (large K_z at each layer) or depth (many layers). At present, this tradeoff is not well understood.[3]

[3] With even a single hidden layer, a neural network can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^N$ to an arbitrarily small non-zero error; see section 6.4.1 of Goodfellow et al. (2016)

It is also possible to "short circuit" a hidden layer, by propagating information directly from the input to the next higher level of the network. This is the idea behind **residual networks**, which propagate information directly from the input to the subsequent layer (He et al., 2016),

$$\boldsymbol{z} = f(\boldsymbol{\Theta}^{(x\rightarrow z)}\boldsymbol{x}) + \boldsymbol{x}, \tag{3.13}$$

where f is any nonlinearity, such as sigmoid or ReLU. A more complex architecture is the **highway network** (Srivastava et al., 2015; Kim et al., 2016), in which an addition **gate** controls an interpolation between $f(\boldsymbol{\Theta}^{(x\rightarrow z)}\boldsymbol{x})$ and $\boldsymbol{x}$,

$$\boldsymbol{t} = \sigma(\Theta^{(t)}\boldsymbol{x} + \boldsymbol{b}^{(t)}) \tag{3.14}$$

$$\boldsymbol{z} = \boldsymbol{t} \odot f(\boldsymbol{\Theta}^{(x\rightarrow z)}\boldsymbol{x}) + (\mathbf{1} - \boldsymbol{t}) \odot \boldsymbol{x}, \tag{3.15}$$

where $\odot$ refers to an elementwise vector product, and $\mathbf{1}$ is a column vector of ones. As before, the sigmoid function is applied elementwise to its input; recall that the output of this function is restricted to the range $(0, 1)$. Gating is also used in the **long short-term memory (LSTM)**, which is discussed in chapter 6. Residual and highway connections address a problem with deep architectures: repeated application of a nonlinear activation function can make it difficult to learn the parameters of the lower levels of the network, which are too distant from the supervision signal.

3.2.3 Outputs and loss functions

In the multi-class classification example, a softmax output produces probabilities over each possible label. This aligns with a negative **conditional log-likelihood**,

$$-\mathcal{L} = -\sum_{i=1}^{N} \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \Theta). \tag{3.16}$$

where $\Theta = \{\boldsymbol{\Theta}^{(x\rightarrow z)}, \boldsymbol{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}\}$ is the entire set of parameters.

This loss can be written alternatively as follows:

$$\tilde{y}_j \triangleq \Pr(y = j \mid \boldsymbol{x}^{(i)}; \Theta) \tag{3.17}$$

$$-\mathcal{L} = -\sum_{i=1}^{N} \boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}} \tag{3.18}$$

for a survey of these theoretical results. However, depending on the function to be approximated, the width of the hidden layer may need to be arbitrarily large. Furthermore, the fact that a network has the *capacity* to approximate any given function does not imply that it is possible to *learn* the function using gradient-based optimization.

where $\boldsymbol{e}_{y^{(i)}}$ is a **one-hot vector** of zeros with a value of 1 at position $y^{(i)}$. The inner product between $\boldsymbol{e}_{y^{(i)}}$ and $\log \tilde{\boldsymbol{y}}$ is also called the multinomial **cross-entropy**, and this terminology is preferred in many neural networks papers and software packages.

It is also possible to train neural networks from other objectives, such as a margin loss. In this case, it is not necessary to use softmax at the output layer: an affine transformation of the hidden layer is enough:

$$\Psi(y; \boldsymbol{x}^{(i)}, \Theta) = \boldsymbol{\theta}_y^{(z \to y)} \cdot \boldsymbol{z} + b_y \quad [3.19]$$

$$\ell_{\text{MARGIN}}(\Theta; \boldsymbol{x}^{(i)}, y^{(i)}) = \max_{y \neq y^{(i)}} \Big(1 + \Psi(y; \boldsymbol{x}^{(i)}, \Theta) - \Psi(y^{(i)}; \boldsymbol{x}^{(i)}, \Theta)\Big)_+ . \quad [3.20]$$

In regression problems, the output is a scalar or vector (see § 4.1.2). For these problems, a typical loss function is the squared error $(y - \hat{y})^2$ or squared norm $||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2^2$.

3.2.4 Inputs and lookup layers

In text classification, the input layer $\boldsymbol{x}$ can refer to a bag-of-words vector, where x_j is the count of word j. The input to the hidden unit z_k is then $\sum_{j=1}^{V} \theta_{j,k}^{(x \to z)} x_j$, and word j is represented by the vector $\boldsymbol{\theta}_j^{(x \to z)}$. This vector is sometimes described as the **embedding** of word j, and can be learned from unlabeled data, using techniques discussed in chapter 14. The columns of $\boldsymbol{\Theta}^{(x \to z)}$ are each K_z-dimensional word embeddings.

Chapter 2 presented an alternative view of text documents, as a sequence of word tokens, $w_1, w_2, \ldots, w_M$. In a neural network, each word token w_m is represented with a one-hot vector, $\boldsymbol{e}_{w_m}$, with dimension V. The matrix-vector product $\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{e}_{w_m}$ returns the embedding of word w_m. The complete document can represented by horizontally concatenating these one-hot vectors, $\mathbf{W} = [\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \ldots, \boldsymbol{e}_{w_M}]$, and the bag-of-words representation can be recovered from the matrix-vector product $\mathbf{W}[1, 1, \ldots, 1]^\top$, which sums each row over the tokens $m = \{1, 2, \ldots, M\}$. The matrix product $\boldsymbol{\Theta}^{(x \to z)} \mathbf{W}$ contains the horizontally concatenated embeddings of each word in the document, which will be useful as the starting point for **convolutional neural networks** (see § 3.4). This is sometimes called a **lookup layer**, because the first step is to lookup the embeddings for each word in the input text.

3.3 Learning neural networks

The feedforward network in Figure 3.1 can now be written as,

$$\boldsymbol{z} \leftarrow f(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x}^{(i)}) \quad [3.21]$$

$$\tilde{\boldsymbol{y}} \leftarrow \text{SoftMax}\Big(\boldsymbol{\Theta}^{(z \to y)} \boldsymbol{z} + \boldsymbol{b}\Big) \quad [3.22]$$

$$\ell^{(i)} \leftarrow - \boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}}, \quad [3.23]$$

where f is an elementwise activation function, such as σ or ReLU, and $\ell^{(i)}$ is the loss at instance i. The parameters $\mathbf{\Theta}^{(x\rightarrow z)}$, $\mathbf{\Theta}^{(z\rightarrow y)}$, and $\boldsymbol{b}$ can be estimated using online gradient-based optimization. The simplest such algorithm is stochastic gradient descent, which was discussed in § 2.6. Each parameter is updated by the gradient of the loss,

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \eta^{(t)} \nabla_{\boldsymbol{b}} \ell^{(i)} \tag{3.24}$$

$$\boldsymbol{\theta}_k^{(z\rightarrow y)} \leftarrow \boldsymbol{\theta}_k^{(z\rightarrow y)} - \eta^{(t)} \nabla_{\boldsymbol{\theta}_k^{(z\rightarrow y)}} \ell^{(i)} \tag{3.25}$$

$$\boldsymbol{\theta}_n^{(x\rightarrow z)} \leftarrow \boldsymbol{\theta}_n^{(x\rightarrow z)} - \eta^{(t)} \nabla_{\boldsymbol{\theta}_n^{(x\rightarrow z)}} \ell^{(i)}, \tag{3.26}$$

where $\eta^{(t)}$ is the learning rate on iteration t, $\ell^{(i)}$ is the loss on instance (or minibatch) i, and $\boldsymbol{\theta}_n^{(x\rightarrow z)}$ is column n of the matrix $\mathbf{\Theta}^{(x\rightarrow z)}$, and $\boldsymbol{\theta}_k^{(z\rightarrow y)}$ is column k of $\mathbf{\Theta}^{(z\rightarrow y)}$.

The gradients of the negative log-likelihood on $\boldsymbol{b}$ and $\boldsymbol{\theta}_k^{(z\rightarrow y)}$ are similar to the gradients in logistic regression. For $\boldsymbol{\theta}^{(z\rightarrow y)}$, the gradient is,

$$\nabla_{\boldsymbol{\theta}_k^{(z\rightarrow y)}} \ell^{(i)} = \left[\frac{\partial \ell^{(i)}}{\partial \theta_{k,1}^{(z\rightarrow y)}}, \frac{\partial \ell^{(i)}}{\partial \theta_{k,2}^{(z\rightarrow y)}}, \ldots, \frac{\partial \ell^{(i)}}{\partial \theta_{k,K_y}^{(z\rightarrow y)}} \right]^\top \tag{3.27}$$

$$\frac{\partial \ell^{(i)}}{\partial \theta_{k,j}^{(z\rightarrow y)}} = -\frac{\partial}{\partial \theta_{k,j}^{(z\rightarrow y)}} \left(\boldsymbol{\theta}_{y^{(i)}}^{(z\rightarrow y)} \cdot \boldsymbol{z} - \log \sum_{y \in \mathcal{Y}} \exp \boldsymbol{\theta}_y^{(z\rightarrow y)} \cdot \boldsymbol{z} \right) \tag{3.28}$$

$$= \Big(\Pr(y = j \mid \boldsymbol{z}; \mathbf{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) - \delta\left(j = y^{(i)}\right) \Big) z_k, \tag{3.29}$$

where $\delta\left(j = y^{(i)}\right)$ is a function that returns one when $j = y^{(i)}$, and zero otherwise. The gradient $\nabla_{\boldsymbol{b}} \ell^{(i)}$ is similar to Equation 3.29.

The gradients on the input layer weights $\mathbf{\Theta}^{(x\rightarrow z)}$ are obtained by the chain rule of differentiation:

$$\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\rightarrow z)}} = \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial z_k}{\partial \theta_{n,k}^{(x\rightarrow z)}} \tag{3.30}$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial f(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x})}{\partial \theta_{n,k}^{(x\rightarrow z)}} \tag{3.31}$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} \times f'(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x}) \times x_n, \tag{3.32}$$

where $f'(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x})$ is the derivative of the activation function f, applied at the input

$\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x}$. For example, if f is the sigmoid function, then the derivative is,

$$\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\rightarrow z)}} = \frac{\partial \ell^{(i)}}{\partial z_k} \times \sigma(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x}) \times (1 - \sigma(\boldsymbol{\theta}_k^{(x\rightarrow z)} \cdot \boldsymbol{x})) \times x_n \quad [3.33]$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} \times z_k \times (1 - z_k) \times x_n. \quad [3.34]$$

For intuition, consider each of the terms in the product.

- If the negative log-likelihood $\ell^{(i)}$ does not depend much on z_k, then $\frac{\partial \ell^{(i)}}{\partial z_k} \approx 0$. In this case it doesn't matter how z_k is computed, and so $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\rightarrow z)}} \approx 0$.
- If z_k is near 1 or 0, then the curve of the sigmoid function is nearly flat (Figure 3.2), and changing the inputs will make little local difference. The term $z_k \times (1 - z_k)$ is maximized at $z_k = \frac{1}{2}$, where the slope of the sigmoid function is steepest.
- If $x_n = 0$, then it does not matter how we set the weights $\theta_{n,k}^{(x\rightarrow z)}$, so $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x\rightarrow z)}} = 0$.

3.3.1 Backpropagation

The equations above rely on the chain rule to compute derivatives of the loss with respect to each parameter of the model. Furthermore, local derivatives are frequently reused: for example, $\frac{\partial \ell^{(i)}}{\partial z_k}$ is reused in computing the derivatives with respect to each $\theta_{n,k}^{(x\rightarrow z)}$. These terms should therefore be computed once, and then cached. Furthermore, we should only compute any derivative once we have already computed all of the necessary "inputs" demanded by the chain rule of differentiation. This combination of sequencing, caching, and differentiation is known as **backpropagation**. It can be generalized to any directed acyclic **computation graph**.

A computation graph is a declarative representation of a computational process. At each node t, compute a value v_t by applying a function f_t to a (possibly empty) list of parent nodes, $\boldsymbol{\pi}_t$. Figure 3.3 shows the computation graph for a feedforward network with one hidden layer. There are nodes for the input $\boldsymbol{x}^{(i)}$, the hidden layer $\boldsymbol{z}$, the predicted output $\hat{\boldsymbol{y}}$, and the parameters Θ. During training, there is also a node for the ground truth label $y^{(i)}$ and the loss $\ell^{(i)}$. The predicted output $\hat{\boldsymbol{y}}$ is one of the parents of the loss (the other is the label $y^{(i)}$); *its* parents include Θ and $\boldsymbol{z}$, and so on.

Computation graphs include three types of nodes:

Variables. In the feedforward network of Figure 3.3, the variables include the inputs $\boldsymbol{x}$, the hidden nodes $\boldsymbol{z}$, the outputs $\boldsymbol{y}$, and the loss function. Inputs are variables that do not have parents. Backpropagation computes the gradients with respect to all

Algorithm 6 General backpropagation algorithm. In the computation graph G, every node contains a function f_t and a set of parent nodes $\boldsymbol{\pi}_t$; the inputs to the graph are $\boldsymbol{x}^{(i)}$.

procedure BACKPROP($G = \{f_t, \boldsymbol{\pi}_t\}_{t=1}^T\}, \boldsymbol{x}^{(i)}$)
  $v_{t(n)} \leftarrow x_n^{(i)}$ for all n and associated computation nodes $t(n)$.
  for $t \in$ TOPOLOGICALSORT(G) **do** ▷ Forward pass: compute value at each node
    if $|\boldsymbol{\pi}_t| > 0$ **then**
      $v_t \leftarrow f_t(v_{\pi_{t,1}}, v_{\pi_{t,2}}, \ldots, v_{\pi_{t,N_t}})$
  $g_{\text{objective}} = 1$ ▷ Backward pass: compute gradients at each node
  for $t \in$ REVERSE(TOPOLOGICALSORT(G)) **do**
    $g_t \leftarrow \sum_{t':t \in \boldsymbol{\pi}_{t'}} g_{t'} \times \nabla_{v_t} v_{t'}$ ▷ Sum over all t' that are children of t, propagating the gradient $g_{t'}$, scaled by the local gradient $\nabla_{v_t} v_{t'}$
  return $\{g_1, g_2, \ldots, g_T\}$

variables except the inputs, and propagates these gradients backwards to the parameters.

Parameters. In a feedforward network, the parameters include the weights and offsets. In Figure 3.3, the parameters are summarized in the node Θ, but we could have separate nodes for $\boldsymbol{\Theta}^{(x \rightarrow z)}$, $\boldsymbol{\Theta}^{(z \rightarrow y)}$, and any offset parameters. Parameter nodes do not have parents; they are not computed from other nodes, but rather, are learned by gradient descent.

Loss. The loss $\ell^{(i)}$ is the quantity that is to be minimized during training. The node representing the loss in the computation graph is not the parent of any other node; its parents are typically the predicted label $\hat{\boldsymbol{y}}$ and the true label $\boldsymbol{y}^{(i)}$. Backpropagation begins by computing the gradient of the loss, and then propagating this gradient backwards to its immediate parents.

If the computation graph is a directed acyclic graph, then it is possible to order the nodes with a topological sort, so that if node t is a parent of node t', then $t < t'$. This means that the values $\{v_t\}_{t=1}^T$ can be computed in a single forward pass. The topological sort is reversed when computing gradients: each gradient g_t is computed from the gradients of the children of t, implementing the chain rule of differentiation. The general backpropagation algorithm for computation graphs is shown in Algorithm 6.

While the gradients with respect to each parameter may be complex, they are composed of products of simple parts. For many networks, all gradients can be computed through **automatic differentiation**. This means that you need only specify the feedforward computation, and the gradients necessary for learning can be obtained automatically. There are many software libraries that perform automatic differentiation on compu-

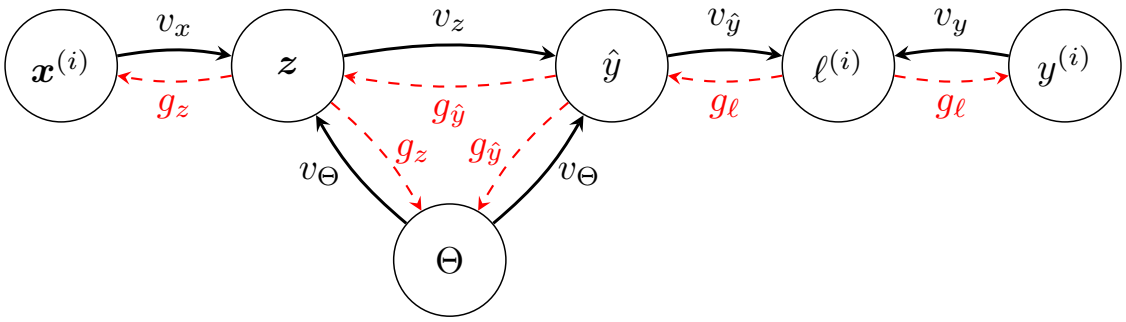


Figure 3.3: A computation graph for the feedforward neural network shown in Figure 3.1.

tation graphs, such as TORCH (Collobert et al., 2011), TENSORFLOW (Abadi et al., 2016), and DYNET (Neubig et al., 2017). One important distinction between these libraries is whether they support **dynamic computation graphs**, in which the structure of the computation graph varies across instances. Static computation graphs are compiled in advance, and can be applied to fixed-dimensional data, such as bag-of-words vectors. In many natural language processing problems, each input has a distinct structure, requiring a unique computation graph. A simple case occurs in recurrent neural network language models (see chapter 6), in which there is one node for each word in a sentence. More complex cases include recursive neural networks (see chapter 14), in which the network is a tree structure matching the syntactic organization of the input.

3.3.2 Regularization and dropout

In linear classification, overfitting was addressed by augmenting the objective with a regularization term, $\lambda||\boldsymbol{\theta}||_2^2$. This same approach can be applied to feedforward neural networks, penalizing each matrix of weights:

$$L = \sum_{i=1}^{N} \ell^{(i)} + \lambda_{z\rightarrow y}||\boldsymbol{\Theta}^{(z\rightarrow y)}||_F^2 + \lambda_{x\rightarrow z}||\boldsymbol{\Theta}^{(x\rightarrow z)}||_F^2, \quad [3.35]$$

where $||\boldsymbol{\Theta}||_F^2 = \sum_{i,j} \theta_{i,j}^2$ is the squared **Frobenius norm**, which generalizes the L_2 norm to matrices. The bias parameters $\boldsymbol{b}$ are not regularized, as they do not contribute to the sensitivity of the classifier to the inputs. In gradient-based optimization, the practical effect of Frobenius norm regularization is that the weights "decay" towards zero at each update, motivating the alternative name **weight decay**.

Another approach to controlling model complexity is **dropout**, which involves randomly setting some computation nodes to zero during training (Srivastava et al., 2014). For example, in the feedforward network, on each training instance, with probability ρ we

set each input x_n and each hidden layer node z_k to zero. Srivastava et al. (2014) recommend $\rho = 0.5$ for hidden units, and $\rho = 0.2$ for input units. Dropout is also incorporated in the gradient computation, so if node z_k is dropped, then none of the weights $\boldsymbol{\theta}_k^{(x \to z)}$ will be updated for this instance. Dropout prevents the network from learning to depend too much on any one feature or hidden node, and prevents **feature co-adaptation**, in which a hidden unit is only useful in combination with one or more other hidden units. Dropout is a special case of **feature noising**, which can also involve adding Gaussian noise to inputs or hidden units (Holmstrom and Koistinen, 1992). Wager et al. (2013) show that dropout is approximately equivalent to "adaptive" L_2 regularization, with a separate regularization penalty for each feature.

3.3.3 *Learning theory

Chapter 2 emphasized the importance of **convexity** for learning: for convex objectives, the global optimum can be found efficiently. The negative log-likelihood and hinge loss are convex functions of the parameters of the output layer. However, the output of a feedforward network is generally not a convex function of the parameters of the input layer, $\boldsymbol{\Theta}^{(x \to z)}$. Feedforward networks can be viewed as function composition, where each layer is a function that is applied to the output of the previous layer. Convexity is generally not preserved in the composition of two convex functions — and furthermore, "squashing" activation functions like tanh and sigmoid are not convex.

The non-convexity of hidden layer neural networks can also be seen by permuting the elements of the hidden layer, from $\boldsymbol{z} = [z_1, z_2, \ldots, z_{K_z}]$ to $\tilde{\boldsymbol{z}} = [z_{\pi(1)}, z_{\pi(2)}, \ldots, z_{\pi(K_z)}]$. This corresponds to applying π to the rows of $\boldsymbol{\Theta}^{(x \to z)}$ and the columns of $\boldsymbol{\Theta}^{(z \to y)}$, resulting in permuted parameter matrices $\boldsymbol{\Theta}_\pi^{(x \to z)}$ and $\boldsymbol{\Theta}_\pi^{(z \to y)}$. As long as this permutation is applied consistently, the loss will be identical, $L(\boldsymbol{\Theta}) = L(\boldsymbol{\Theta}_\pi)$: it is *invariant* to this permutation. However, the loss of the linear combination $L(\alpha\boldsymbol{\Theta} + (1-\alpha)\boldsymbol{\Theta}_\pi)$ will generally not be identical to the loss under $\boldsymbol{\Theta}$ or its permutations. If $L(\boldsymbol{\Theta})$ is better than the loss at any points in the immediate vicinity, and if $L(\boldsymbol{\Theta}) = L(\boldsymbol{\Theta}_\pi)$, then the loss function does not satisfy the definition of convexity (see § 2.4). One of the exercises asks you to prove this more rigorously.

In practice, the existence of multiple optima is not necessary problematic, if all such optima are permutations of the sort described in the previous paragraph. In contrast, "bad" local optima are better than their neighbors, but much worse than the global optimum. Fortunately, in large feedforward neural networks, most local optima are nearly as good as the global optimum (Choromanska et al., 2015). More generally, a **critical point** is one at which the gradient is zero. Critical points may be local optima, but they may also be **saddle points**, which are local minima in some directions, but local *maxima* in other directions. For example, the equation $x_1^2 - x_2^2$ has a saddle point at $\boldsymbol{x} = (0, 0)$. In large networks, the overwhelming majority of critical points are saddle points, rather

than local minima or maxima (Dauphin et al., 2014). Saddle points can pose problems for gradient-based optimization, since learning will slow to a crawl as the gradient goes to zero. However, the noise introduced by stochastic gradient descent, and by feature noising techniques such as dropout, can help online optimization to escape saddle points and find high-quality optima (Ge et al., 2015). Other techniques address saddle points directly, using local reconstructions of the Hessian matrix (Dauphin et al., 2014) or higher-order derivatives (Anandkumar and Ge, 2016).

Another theoretical puzzle about neural networks is how they are able to **generalize** to unseen data. Given enough parameters, a two-layer feedforward network can "memorize" its training data, attaining perfect accuracy on any training set. A particularly salient demonstration was provided by Zhang et al. (2017), who showed that neural networks can learn to perfectly classify a training set of images, even when the labels are replaced with random values! Of course, this network attains only chance accuracy when applied to heldout data. The concern is that when such a powerful learner is applied to real training data, it may learn a pathological classification function, which exploits irrelevant details of the training data and fails to generalize. Yet this extreme **overfitting** is rarely encountered in practice, and can usually be prevented by regularization, dropout, and early stopping (see § 3.3.4). Recent papers have derived generalization guarantees for specific classes of neural networks (e.g., Kawaguchi et al., 2017; Brutzkus et al., 2018), but theoretical work in this area is ongoing.

3.3.4 Tricks

Getting neural networks to work sometimes requires heuristic "tricks" (Bottou, 2012; Goodfellow et al., 2016; Goldberg, 2017b). This section presents some tricks that are especially important.

Initialization Initialization is not especially important for linear classifiers, since convexity ensures that the global optimum can usually be found quickly. But for multilayer neural networks, it is helpful to have a good starting point. One reason is that if the magnitude of the initial weights is too large, a sigmoid or tanh nonlinearity will be saturated, leading to a small gradient, and slow learning. Large gradients can cause training to diverge, with the parameters taking increasingly extreme values until reaching the limits of the floating point representation.

Initialization can help avoid these problems by ensuring that the variance over the initial gradients is constant and bounded throughout the network. For networks with tanh activation functions, this can be achieved by sampling the initial weights from the

following uniform distribution (Glorot and Bengio, 2010),

$$\theta_{i,j} \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_{\text{in}}(n)+d_{\text{out}}(n)}}, \frac{\sqrt{6}}{\sqrt{d_{\text{in}}(n)+d_{\text{out}}(n)}}\right], \tag{3.36}$$

$$\tag{3.37}$$

For the weights leading to a ReLU activation function, He et al. (2015) use similar argumentation to justify sampling from a zero-mean Gaussian distribution,

$$\theta_{i,j} \sim N(0, \sqrt{2/d_{\text{in}}(n)}) \tag{3.38}$$

Rather than initializing the weights independently, it can be beneficial to initialize each layer jointly as an **orthonormal matrix**, ensuring that $\boldsymbol{\Theta}^\top\boldsymbol{\Theta} = \mathbb{I}$ (Saxe et al., 2014). Orthonormal matrices preserve the norm of the input, so that $||\boldsymbol{\Theta}\boldsymbol{x}|| = ||\boldsymbol{x}||$, which prevents the gradients from exploding or vanishing. Orthogonality ensures that the hidden units are uncorrelated, so that they correspond to different features of the input. Orthonormal initialization can be performed by applying **singular value decomposition** to a matrix of values sampled from a standard normal distribution:

$$a_{i,j} \sim N(0,1) \tag{3.39}$$

$$\mathbf{A} = \{a_{i,j}\}_{i=1,j=1}^{d_{\text{in}}(j),d_{\text{out}}(j)} \tag{3.40}$$

$$\mathbf{U}, \mathbf{S}, \mathbf{V}^\top = \text{SVD}(\mathbf{A}) \tag{3.41}$$

$$\boldsymbol{\Theta}^{(j)} \leftarrow \mathbf{U}. \tag{3.42}$$

The matrix $\mathbf{U}$ contains the **singular vectors** of $\mathbf{A}$, and is guaranteed to be orthonormal. For more on singular value decomposition, see chapter 14.

Even with careful initialization, there can still be significant variance in the final results. It can be useful to make multiple training runs, and select the one with the best performance on a heldout development set.

Clipping and normalization Learning can be sensitive to the magnitude of the gradient: too large, and learning can diverge, with successive updates thrashing between increasingly extreme values; too small, and learning can grind to a halt. Several heuristics have been proposed to address this issue.

- In **gradient clipping** (Pascanu et al., 2013), an upper limit is placed on the norm of the gradient, and the gradient is rescaled when this limit is exceeded,

$$\text{CLIP}(\tilde{\boldsymbol{g}}) = \begin{cases} \boldsymbol{g} & ||\hat{\boldsymbol{g}}|| < \tau \\ \frac{\tau}{||\boldsymbol{g}||}\boldsymbol{g} & \text{otherwise.} \end{cases} \tag{3.43}$$

- In **batch normalization** (Ioffe and Szegedy, 2015), the inputs to each computation node are recentered by their mean and variance across all of the instances in the minibatch $\mathcal{B}$ (see § 2.6.2). For example, in a feedforward network with one hidden layer, batch normalization would tranform the inputs to the hidden layer as follows:

$$\boldsymbol{\mu}^{(\mathcal{B})} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \boldsymbol{x}^{(i)} \tag{3.44}$$

$$\boldsymbol{s}^{(\mathcal{B})} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(\mathcal{B})})^2 \tag{3.45}$$

$$\overline{\boldsymbol{x}}^{(i)} = (\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(\mathcal{B})}) / \sqrt{\boldsymbol{s}^{(\mathcal{B})}}. \tag{3.46}$$

  Empirically, this speeds convergence of deep architectures. One explanation is that it helps to correct for changes in the distribution of activations during training.

- In **layer normalization** (Ba et al., 2016), the inputs to each nonlinear activation function are recentered across the layer:

$$\boldsymbol{a} = \boldsymbol{\Theta}^{(x \rightarrow z)} \boldsymbol{x} \tag{3.47}$$

$$\mu = \frac{1}{K_z} \sum_{k=1}^{K_z} a_k \tag{3.48}$$

$$s = \frac{1}{K_z} \sum_{k=1}^{K_z} (a_k - \mu)^2 \tag{3.49}$$

$$\boldsymbol{z} = (\boldsymbol{a} - \mu) / \sqrt{s}. \tag{3.50}$$

  Layer normalization has similar motivations to batch normalization, but it can be applied across a wider range of architectures and training conditions.

Online optimization There is a cottage industry of online optimization algorithms that attempt to improve on stochastic gradient descent. **AdaGrad** was reviewed in § 2.6.2; its main innovation is to set adaptive learning rates for each parameter by storing the sum of squared gradients. Rather than using the sum over the entire training history, we can keep a running estimate,

$$v_j^{(t)} = \beta v_j^{(t-1)} + (1 - \beta) g_{t,j}^2, \tag{3.51}$$

where $g_{t,j}$ is the gradient with respect to parameter j at time t, and $\beta \in [0, 1]$. This term places more emphasis on recent gradients, and is employed in the AdaDelta (Zeiler, 2012) and Adam (Kingma and Ba, 2014) optimizers. Online optimization and its theoretical background are reviewed by Bottou et al. (2016). **Early stopping**, mentioned in § 2.3.2, can help to avoid overfitting by terminating training after reaching a plateau in the performance on a heldout validation set.

Practical advice The bag of tricks for training neural networks continues to grow, and it is likely that there will be several new ones by the time you read this. Today, it is standard practice to use gradient clipping, early stopping, and a sensible initialization of parameters to small random values. More bells and whistles can be added as solutions to specific problems — for example, if it is difficult to find a good learning rate for stochastic gradient descent, then it may help to try a fancier optimizer with an adaptive learning rate. Alternatively, if a method such as layer normalization is used by related models in the research literature, you should probably consider it, especially if you are having trouble matching published results. As with linear classifiers, it is important to evaluate these decisions on a held-out development set, and not on the test set that will be used to provide the final measure of the model's performance (see § 2.2.5).

3.4 Convolutional neural networks

A basic weakness of the bag-of-words model is its inability to account for the ways in which words combine to create meaning, including even simple reversals such as *not pleasant*, *hardly a generous offer*, and *I wouldn't mind missing the flight*. Computer vision faces the related challenge of identifying the semantics of images from pixel features that are uninformative in isolation. An earlier generation of computer vision research focused on designing *filters* to aggregate local pixel-level features into more meaningful representations, such as edges and corners (e.g., Canny, 1987). Similarly, earlier NLP research attempted to capture multiword linguistic phenomena by hand-designed lexical patterns (Hobbs et al., 1997). In both cases, the output of the filters and patterns could then act as base features in a linear classifier. But rather than designing these feature extractors by hand, a better approach is to learn them, using the magic of backpropagation. This is the idea behind **convolutional neural networks**.

Following § 3.2.4, define the base layer of a neural network as,

$$\mathbf{X}^{(0)} = \mathbf{\Theta}^{(x \rightarrow z)}[\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \ldots, \boldsymbol{e}_{w_M}], \tag{3.52}$$

where $\boldsymbol{e}_{w_m}$ is a column vector of zeros, with a 1 at position w_m. The base layer has dimension $\mathbf{X}^{(0)} \in \mathbb{R}^{K_e \times M}$, where K_e is the size of the word embeddings. To merge information across adjacent words, we *convolve* $\mathbf{X}^{(0)}$ with a set of filter matrices $\mathbf{C}^{(k)} \in \mathbb{R}^{K_e \times h}$. Convolution is indicated by the symbol $*$, and is defined,

$$\mathbf{X}^{(1)} = f(\boldsymbol{b} + \mathbf{C} * \mathbf{X}^{(0)}) \quad \Longrightarrow \quad x^{(1)}_{k,m} = f\left(b_k + \sum_{k'=1}^{K_e} \sum_{n=1}^{h} c^{(k)}_{k',n} \times x^{(0)}_{k',m+n-1}\right), \tag{3.53}$$

where f is an activation function such as tanh or ReLU, and $\boldsymbol{b}$ is a vector of offsets. The convolution operation slides the matrix $\mathbf{C}^{(k)}$ across the columns of $\mathbf{X}^{(0)}$. At each position m, we compute the elementwise product $\mathbf{C}^{(k)} \odot \mathbf{X}^{(0)}_{m:m+h-1}$, and take the sum.

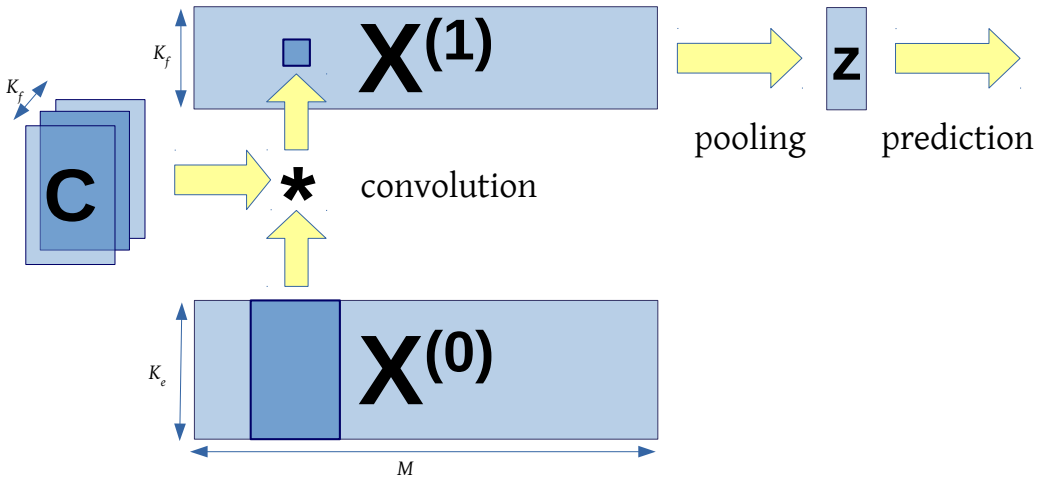


Figure 3.4: A convolutional neural network for text classification

A simple filter might compute a weighted average over nearby words,

$$\mathbf{C}^{(k)} = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.5 & 1 & 0.5 \\ \dots & \dots & \dots \\ 0.5 & 1 & 0.5 \end{bmatrix}, \quad [3.54]$$

thereby representing trigram units like *not so unpleasant*. In **one-dimensional convolution**, each filter matrix $\mathbf{C}^{(k)}$ is constrained to have non-zero values only at row k (Kalchbrenner et al., 2014). This means that each dimension of the word embedding is processed by a separate filter, and it implies that $K_f = K_e$.

To deal with the beginning and end of the input, the base matrix $\mathbf{X}^{(0)}$ may be padded with h column vectors of zeros at the beginning and end; this is known as **wide convolution**. If padding is not applied, then the output from each layer will be $h - 1$ units smaller than the input; this is known as **narrow convolution**. The filter matrices need not have identical filter widths, so more generally we could write h_k to indicate to width of filter $\mathbf{C}^{(k)}$. As suggested by the notation $\mathbf{X}^{(0)}$, multiple layers of convolution may be applied, so that $\mathbf{X}^{(d)}$ is the input to $\mathbf{X}^{(d+1)}$.

After D convolutional layers, we obtain a matrix representation of the document $\mathbf{X}^{(D)} \in \mathbb{R}^{K_z \times M}$. If the instances have variable lengths, it is necessary to aggregate over all M word positions to obtain a fixed-length representation. This can be done by a **pooling** operation,

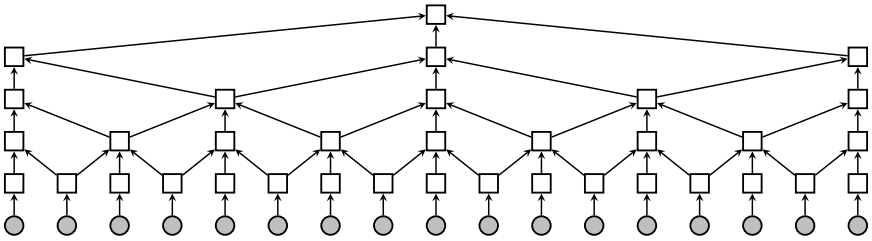


Figure 3.5: A dilated convolutional neural network captures progressively larger context through recursive application of the convolutional operator

such as max-pooling (Collobert et al., 2011) or average-pooling,

$$\boldsymbol{z} = \text{MaxPool}(\mathbf{X}^{(D)}) \quad \Longrightarrow \quad z_k = \max\left(x_{k,1}^{(D)}, x_{k,2}^{(D)}, \ldots x_{k,M}^{(D)}\right) \tag{3.55}$$

$$\boldsymbol{z} = \text{AvgPool}(\mathbf{X}^{(D)}) \quad \Longrightarrow \quad z_k = \frac{1}{M}\sum_{m=1}^{M} x_{k,m}^{(D)}. \tag{3.56}$$

The vector $\boldsymbol{z}$ can now act as a layer in a feedforward network, culminating in a prediction $\hat{y}$ and a loss $\ell^{(i)}$. The setup is shown in Figure 3.4.

Just as in feedforward networks, the parameters $(\mathbf{C}^{(k)}, \boldsymbol{b}, \Theta)$ can be learned by backpropagating from the classification loss. This requires backpropagating through the max-pooling operation, which is a discontinuous function of the input. But because we need only a local gradient, backpropagation flows only through the argmax m:

$$\frac{\partial z_k}{\partial x_{k,m}^{(D)}} = \begin{cases} 1, & x_{k,m}^{(D)} = \max\left(x_{k,1}^{(D)}, x_{k,2}^{(D)}, \ldots x_{k,M}^{(D)}\right) \\ 0, & \text{otherwise.} \end{cases} \tag{3.57}$$

The computer vision literature has produced a huge variety of convolutional architectures, and many of these innovations can be applied to text data. One avenue for improvement is more complex pooling operations, such as k-max pooling (Kalchbrenner et al., 2014), which returns a matrix of the k largest values for each filter. Another innovation is the use of **dilated convolution** to build multiscale representations (Yu and Koltun, 2016). At each layer, the convolutional operator applied in *strides*, skipping ahead by s steps after each feature. As we move up the hierarchy, each layer is s times smaller than the layer below it, effectively summarizing the input (Kalchbrenner et al., 2016; Strubell et al., 2017). This idea is shown in Figure 3.5. Multi-layer convolutional networks can also be augmented with "shortcut" connections, as in the residual network from § 3.2.2 (Johnson and Zhang, 2017).

Additional resources

The deep learning textbook by Goodfellow et al. (2016) covers many of the topics in this chapter in more detail. For a comprehensive review of neural networks in natural language processing, see Goldberg (2017b). A seminal work on deep learning in natural language processing is the aggressively titled "Natural Language Processing (Almost) from Scratch", which uses convolutional neural networks to perform a range of language processing tasks (Collobert et al., 2011), although there is earlier work (e.g., Henderson, 2004). This chapter focuses on feedforward and convolutional neural networks, but recurrent neural networks are one of the most important deep learning architectures for natural language processing. They are covered extensively in chapters 6 and 7.

The role of deep learning in natural language processing research has caused angst in some parts of the natural language processing research community (e.g., Goldberg, 2017a), especially as some of the more zealous deep learning advocates have argued that end-to-end learning from "raw" text can eliminate the need for linguistic constructs such as sentences, phrases, and even words (Zhang et al., 2015, originally titled "Text understanding from scratch"). These developments were surveyed by Manning (2015). While reports of the demise of linguistics in natural language processing remain controversial at best, deep learning and backpropagation have become ubiquitous in both research and applications.

Exercises

1. Figure 3.3 shows the computation graph for a feedforward neural network with one layer.

   a) Update the computation graph to include a residual connection between $\boldsymbol{x}$ and $\boldsymbol{z}$.

   b) Update the computation graph to include a highway connection between $\boldsymbol{x}$ and $\boldsymbol{z}$.

2. Prove that the softmax and sigmoid functions are equivalent when the number of possible labels is two. Specifically, for any $\boldsymbol{\Theta}^{(z \rightarrow y)}$ (omitting the offset $\boldsymbol{b}$ for simplicity), show how to construct a vector of weights $\boldsymbol{\theta}$ such that,

$$\text{SoftMax}(\boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z})[0] = \sigma(\boldsymbol{\theta} \cdot \boldsymbol{z}). \quad [3.58]$$

3. Convolutional neural networks often aggregate across words by using max-**pooling** (Equation 3.55 in § 3.4). A potential concern is that there is zero gradient with respect to the parts of the input that are not included in the maximum. The following

questions consider the gradient with respect to an element of the input, $x_{m,k}^{(0)}$, and they assume that all parameters are independently distributed.

a) First consider a minimal network, with $\boldsymbol{z} = \text{MaxPool}(\mathbf{X}^{(0)})$. What is the probability that the gradient $\frac{\partial \ell}{\partial x_{m,k}^{(0)}}$ is non-zero?

b) Now consider a two-level network, with $\mathbf{X}^{(1)} = f(\boldsymbol{b} + \mathbf{C} * \mathbf{X}^{(0)})$. Express the probability that the gradient $\frac{\partial \ell}{\partial x_{m,k}^{(0)}}$ is non-zero, in terms of the input length M, the filter size n, and the number of filters K_f.

c) Using a calculator, work out the probability for the case $M = 128, n = 4, K_f = 32$.

d) Now consider a three-level network, $\mathbf{X}^{(2)} = f(\boldsymbol{b} + \mathbf{C} * \mathbf{X}^{(1)})$. Give the general equation for the probability that $\frac{\partial \ell}{\partial x_{m,k}^{(0)}}$ is non-zero, and compute the numerical probability for the scenario in the previous part, assuming $K_f = 32$ and $n = 4$ at both levels.

4. Design a feedforward network to compute the XOR function:

$$f(x_1, x_2) = \begin{cases} -1, & x_1 = 1, x_2 = 1 \\ 1, & x_1 = 1, x_2 = 0 \\ 1, & x_1 = 0, x_2 = 1 \\ -1, & x_1 = 0, x_2 = 0 \end{cases}. \quad [3.59]$$

Your network should have a single output node which uses the Sign activation function, $f(x) = \begin{cases} 1, & x > 0 \\ -1, & x \leq 0. \end{cases}$. Use a single hidden layer, with ReLU activation functions. Describe all weights and offsets.

5. Consider the same network as above (with ReLU activations for the hidden layer), with an arbitrary differentiable loss function $\ell(y^{(i)}, \tilde{y})$, where $\tilde{y}$ is the activation of the output node. Suppose all weights and offsets are initialized to zero. Show that gradient descent will not learn the desired function from this initialization.

6. The simplest solution to the previous problem relies on the use of the ReLU activation function at the hidden layer. Now consider a network with arbitrary activations on the hidden layer. Show that if the initial weights are any uniform constant, then gradient descent will not learn the desired function from this initialization.

7. Consider a network in which: the base features are all binary, $\boldsymbol{x} \in \{0, 1\}^M$; the hidden layer activation function is sigmoid, $z_k = \sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x})$; and the initial weights are sampled independently from a standard normal distribution, $\theta_{j,k} \sim N(0, 1)$.

- Show how the probability of a small initial gradient on any weight, $\frac{\partial z_k}{\partial \boldsymbol{\theta}_{j,k}} < \alpha$, depends on the size of the input M. **Hint**: use the lower bound,

$$\Pr(\sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x}) \times (1 - \sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x})) < \alpha) \quad \geq \quad 2\Pr(\sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x}) < \alpha), \tag{3.60}$$

  and relate this probability to the variance $V[\boldsymbol{\theta}_k \cdot \boldsymbol{x}]$.
- Design an alternative initialization that removes this dependence.

8. The ReLU activation function can lead to "dead neurons", which can never be activated on any input. Consider the following two-layer feedforward network with a scalar output y:

$$z_i = \text{ReLU}(\boldsymbol{\theta}_i^{(x \to z)} \cdot \boldsymbol{x} + b_i) \tag{3.61}$$
$$y = \boldsymbol{\theta}^{(z \to y)} \cdot \boldsymbol{z}. \tag{3.62}$$

   Suppose that the input is a binary vector of observations, $\boldsymbol{x} \in \{0, 1\}^D$.

   a) Under what condition is node z_i "dead"? Your answer should be expressed in terms of the parameters $\boldsymbol{\theta}_i^{(x \to z)}$ and b_i.
   b) Suppose that the gradient of the loss on a given instance is $\frac{\partial \ell}{\partial y} = 1$. Derive the gradients $\frac{\partial \ell}{\partial b_i}$ and $\frac{\partial \ell}{\partial \theta_{j,i}^{(x \to z)}}$ for such an instance.
   c) Using your answers to the previous two parts, explain why a dead neuron can never be brought back to life during gradient-based learning.

9. Suppose that the parameters $\Theta = \{\Theta^{(x \to z)}, \Theta(z \to y), \boldsymbol{b}\}$ are a local optimum of a feedforward network in the following sense: there exists some $\epsilon > 0$ such that,

$$\Big(||\tilde{\Theta}^{(x \to z)} - \Theta^{(x \to z)}||_F^2 + ||\tilde{\Theta}^{(z \to y)} - \Theta^{(z \to y)}||_F^2 + ||\tilde{\boldsymbol{b}} - \boldsymbol{b}||_2^2 < \epsilon\Big)$$
$$\Rightarrow \Big(L(\tilde{\Theta}) > L(\Theta)\Big) \tag{3.63}$$

   Define the function π as a permutation on the hidden units, as described in § 3.3.3, so that for any Θ, $L(\Theta) = L(\Theta_\pi)$. Prove that if a feedforward network has a local optimum in the sense of Equation 3.63, then its loss is not a convex function of the parameters Θ, using the definition of convexity from § 2.4

10. Consider a network with a single hidden layer, and a single output,

$$y = \boldsymbol{\theta}^{(z \to y)} \cdot g(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x}). \tag{3.64}$$

    Assume that g is the ReLU function. Show that for any matrix of weights $\boldsymbol{\Theta}^{(x \to z)}$, it is permissible to rescale each row to have a norm of one, because an identical output can be obtained by finding a corresponding rescaling of $\boldsymbol{\theta}^{(z \to y)}$.

Chapter 4

Linguistic applications of classification

Having covered several techniques for classification, this chapter shifts the focus from mathematics to linguistic applications. Later in the chapter, we will consider the design decisions involved in text classification, as well as best practices for evaluation.

4.1 Sentiment and opinion analysis

A popular application of text classification is to automatically determine the **sentiment** or **opinion polarity** of documents such as product reviews and social media posts. For example, marketers are interested to know how people respond to advertisements, services, and products (Hu and Liu, 2004); social scientists are interested in how emotions are affected by phenomena such as the weather (Hannak et al., 2012), and how both opinions and emotions spread over social networks (Coviello et al., 2014; Miller et al., 2011). In the field of **digital humanities**, literary scholars track plot structures through the flow of sentiment across a novel (Jockers, 2015).[1]

Sentiment analysis can be framed as a direct application of document classification, assuming reliable labels can be obtained. In the simplest case, sentiment analysis is a two or three-class problem, with sentiments of POSITIVE, NEGATIVE, and possibly NEUTRAL. Such annotations could be annotated by hand, or obtained automatically through a variety of means:

- Tweets containing happy emoticons can be marked as positive, sad emoticons as negative (Read, 2005; Pak and Paroubek, 2010).

[1] Comprehensive surveys on sentiment analysis and related problems are offered by Pang and Lee (2008) and Liu (2015).

- Reviews with four or more stars can be marked as positive, three or fewer stars as negative (Pang et al., 2002).
- Statements from politicians who are voting for a given bill are marked as positive (towards that bill); statements from politicians voting against the bill are marked as negative (Thomas et al., 2006).

The bag-of-words model is a good fit for sentiment analysis at the document level: if the document is long enough, we would expect the words associated with its true sentiment to overwhelm the others. Indeed, **lexicon-based sentiment analysis** avoids machine learning altogether, and classifies documents by counting words against positive and negative sentiment word lists (Taboada et al., 2011).

Lexicon-based classification is less effective for short documents, such as single-sentence reviews or social media posts. In these documents, linguistic issues like **negation** and **irrealis** (Polanyi and Zaenen, 2006) — events that are hypothetical or otherwise non-factual — can make bag-of-words classification ineffective. Consider the following examples:

(4.1) a. That's not bad for the first day.
b. This is not the worst thing that can happen.
c. It would be nice if you acted like you understood.
d. There is no reason at all to believe that the polluters are suddenly going to become reasonable. (Wilson et al., 2005)
e. This film should be brilliant. The actors are first grade. Stallone plays a happy, wonderful man. His sweet wife is beautiful and adores him. He has a fascinating gift for living life fully. It sounds like a great plot, **however**, the film is a failure. (Pang et al., 2002)

A minimal solution is to move from a bag-of-words model to a bag-of-**bigrams** model, where each base feature is a pair of adjacent words, e.g.,

$$(\textit{that's}, \textit{not}), (\textit{not}, \textit{bad}), (\textit{bad}, \textit{for}), \ldots \quad [4.1]$$

Bigrams can handle relatively straightforward cases, such as when an adjective is immediately negated; trigrams would be required to extend to larger contexts (e.g., *not the worst*). But this approach will not scale to more complex examples like (4.1d) and (4.1e). More sophisticated solutions try to account for the syntactic structure of the sentence (Wilson et al., 2005; Socher et al., 2013), or apply more complex classifiers such as convolutional neural networks (Kim, 2014), which are described in chapter 3.

4.1.1 Related problems

Subjectivity Closely related to sentiment analysis is **subjectivity detection**, which requires identifying the parts of a text that express subjective opinions, as well as other non-

factual content such as speculation and hypotheticals (Riloff and Wiebe, 2003). This can be done by treating each sentence as a separate document, and then applying a bag-of-words classifier: indeed, Pang and Lee (2004) do exactly this, using a training set consisting of (mostly) subjective sentences gathered from movie reviews, and (mostly) objective sentences gathered from plot descriptions. They augment this bag-of-words model with a graph-based algorithm that encourages nearby sentences to have the same subjectivity label.

Stance classification In debates, each participant takes a side: for example, advocating for or against proposals like adopting a vegetarian lifestyle or mandating free college education. The problem of stance classification is to identify the author's position from the text of the argument. In some cases, there is training data available for each position, so that standard document classification techniques can be employed. In other cases, it suffices to classify each document as whether it is in support or opposition of the argument advanced by a previous document (Anand et al., 2011). In the most challenging case, there is no labeled data for any of the stances, so the only possibility is group documents that advocate the same position (Somasundaran and Wiebe, 2009). This is a form of **unsupervised learning**, discussed in chapter 5.

Targeted sentiment analysis The expression of sentiment is often more nuanced than a simple binary label. Consider the following examples:

(4.2) a. The vodka was good, but the meat was rotten.
b. Go to Heaven for the climate, Hell for the company. –*Mark Twain*

These statements display a mixed overall sentiment: positive towards some entities (e.g., *the vodka*), negative towards others (e.g., *the meat*). **Targeted sentiment analysis** seeks to identify the writer's sentiment towards specific entities (Jiang et al., 2011). This requires identifying the entities in the text and linking them to specific sentiment words — much more than we can do with the classification-based approaches discussed thus far. For example, Kim and Hovy (2006) analyze sentence-internal structure to determine the topic of each sentiment expression.

Aspect-based opinion mining seeks to identify the sentiment of the author of a review towards predefined aspects such as PRICE and SERVICE, or, in the case of (4.2b), CLIMATE and COMPANY (Hu and Liu, 2004). If the aspects are not defined in advance, it may again be necessary to employ unsupervised learning methods to identify them (e.g., Branavan et al., 2009).

Emotion classification While sentiment analysis is framed in terms of positive and negative categories, psychologists generally regard **emotion** as more multifaceted. For example, Ekman (1992) argues that there are six basic emotions — happiness, surprise, fear,

sadness, anger, and contempt — and that they are universal across human cultures. Alm et al. (2005) build a linear classifier for recognizing the emotions expressed in children's stories. The ultimate goal of this work was to improve text-to-speech synthesis, so that stories could be read with intonation that reflected the emotional content. They used bag-of-words features, as well as features capturing the story type (e.g., jokes, folktales), and structural features that reflect the position of each sentence in the story. The task is difficult: even human annotators frequently disagreed with each other, and the best classifiers achieved accuracy between 60-70%.

4.1.2 Alternative approaches to sentiment analysis

Regression A more challenging version of sentiment analysis is to determine not just the class of a document, but its rating on a numerical scale (Pang and Lee, 2005). If the scale is continuous, it is most natural to apply **regression**, identifying a set of weights $\boldsymbol{\theta}$ that minimize the squared error of a predictor $\hat{y} = \boldsymbol{\theta} \cdot \boldsymbol{x} + b$, where b is an offset. This approach is called **linear regression**, and sometimes **least squares**, because the regression coefficients $\boldsymbol{\theta}$ are determined by minimizing the squared error, $(y - \hat{y})^2$. If the weights are regularized using a penalty $\lambda||\boldsymbol{\theta}||_2^2$, then it is **ridge regression**. Unlike logistic regression, both linear regression and ridge regression can be solved in closed form as a system of linear equations.

Ordinal ranking In many problems, the labels are ordered but discrete: for example, product reviews are often integers on a scale of $1 - 5$, and grades are on a scale of $A - F$. Such problems can be solved by discretizing the score $\boldsymbol{\theta} \cdot \boldsymbol{x}$ into "ranks",

$$\hat{y} = \underset{r:\, \boldsymbol{\theta}\cdot\boldsymbol{x} \geq b_r}{\operatorname{argmax}}\, r, \qquad [4.2]$$

where $\boldsymbol{b} = [b_1 = -\infty, b_2, b_3, \ldots, b_K]$ is a vector of boundaries. It is possible to learn the weights and boundaries simultaneously, using a perceptron-like algorithm (Crammer and Singer, 2001).

Lexicon-based classification Sentiment analysis is one of the only NLP tasks where hand-crafted feature weights are still widely employed. In **lexicon-based classification** (Taboada et al., 2011), the user creates a list of words for each label, and then classifies each document based on how many of the words from each list are present. In our linear classification framework, this is equivalent to choosing the following weights:

$$\theta_{y,j} = \begin{cases} 1, & j \in \mathcal{L}_y \\ 0, & \text{otherwise}, \end{cases} \qquad [4.3]$$

where $\mathcal{L}_y$ is the lexicon for label y. Compared to the machine learning classifiers discussed in the previous chapters, lexicon-based classification may seem primitive. However, supervised machine learning relies on large annotated datasets, which are time-consuming and expensive to produce. If the goal is to distinguish two or more categories in a new domain, it may be simpler to start by writing down a list of words for each category.

An early lexicon was the *General Inquirer* (Stone, 1966). Today, popular sentiment lexicons include SENTIWORDNET (Esuli and Sebastiani, 2006) and an evolving set of lexicons from Liu (2015). For emotions and more fine-grained analysis, *Linguistic Inquiry and Word Count* (LIWC) provides a set of lexicons (Tausczik and Pennebaker, 2010). The MPQA lexicon indicates the polarity (positive or negative) of 8221 terms, as well as whether they are strongly or weakly subjective (Wiebe et al., 2005). A comprehensive comparison of sentiment lexicons is offered by Ribeiro et al. (2016). Given an initial **seed lexicon**, it is possible to automatically expand the lexicon by looking for words that frequently co-occur with words in the seed set (Hatzivassiloglou and McKeown, 1997; Qiu et al., 2011).

4.2 Word sense disambiguation

Consider the the following headlines:

(4.3) a. Iraqi head seeks arms
b. Prostitutes appeal to Pope
c. Drunk gets nine years in violin case[2]

These headlines are ambiguous because they contain words that have multiple meanings, or **senses**. Word sense disambiguation is the problem of identifying the intended sense of each word token in a document. Word sense disambiguation is part of a larger field of research called **lexical semantics**, which is concerned with meanings of the words.

At a basic level, the problem of word sense disambiguation is to identify the correct sense for each word token in a document. Part-of-speech ambiguity (e.g., noun versus verb) is usually considered to be a different problem, to be solved at an earlier stage. From a linguistic perspective, senses are not properties of words, but of **lemmas**, which are canonical forms that stand in for a set of inflected words. For example, *arm*/N is a lemma that includes the inflected form *arms*/N — the /N indicates that it we are referring to the noun, and not its **homonym** *arm*/V, which is another lemma that includes the inflected verbs (*arm*/V, *arms*/V, *armed*/V, *arming*/V). Therefore, word sense disambiguation requires first identifying the correct part-of-speech and lemma for each token,

[2]These examples, and many more, can be found at `http://www.ling.upenn.edu/~beatrice/humor/headlines.html`

and then choosing the correct sense from the inventory associated with the corresponding lemma.[3] (Part-of-speech tagging is discussed in § 8.1.)

4.2.1 How many word senses?

Words sometimes have many more than two senses, as exemplified by the word *serve*:

- [FUNCTION]: *The tree stump served as a table*
- [CONTRIBUTE TO]: *His evasive replies only served to heighten suspicion*
- [PROVIDE]: *We serve only the rawest fish*
- [ENLIST]: *She served in an elite combat unit*
- [JAIL]: *He served six years for a crime he didn't commit*
- [LEGAL]: *They were served with subpoenas*[4]

These sense distinctions are annotated in **WORDNET** (`http://wordnet.princeton.edu`), a lexical semantic database for English. WORDNET consists of roughly 100,000 **synsets**, which are groups of lemmas (or phrases) that are synonymous. An example synset is $\{chump^1, fool^2, sucker^1, mark^9\}$, where the superscripts index the sense of each lemma that is included in the synset: for example, there are at least eight other senses of *mark* that have different meanings, and are not part of this synset. A lemma is **polysemous** if it participates in multiple synsets.

WORDNET defines the scope of the word sense disambiguation problem, and, more generally, formalizes lexical semantic knowledge of English. (WordNets have been created for a few dozen other languages, at varying levels of detail.) Some have argued that WordNet's sense granularity is too fine (Ide and Wilks, 2006); more fundamentally, the premise that word senses can be differentiated in a task-neutral way has been criticized as linguistically naïve (Kilgarriff, 1997). One way of testing this question is to ask whether people tend to agree on the appropriate sense for example sentences: according to Mihalcea et al. (2004), people agree on roughly 70% of examples using WordNet senses; far better than chance, but less than agreement on other tasks, such as sentiment annotation (Wilson et al., 2005).

***Other lexical semantic relations** Besides **synonymy**, WordNet also describes many other lexical semantic relationships, including:

- **antonymy**: x means the opposite of y, e.g. FRIEND-ENEMY;

[3] Navigli (2009) provides a survey of approaches for word-sense disambiguation.

[4] Several of the examples are adapted from WORDNET (Fellbaum, 2010).

- **hyponymy**: x is a special case of y, e.g. RED-COLOR; the inverse relationship is **hypernymy**;
- **meronymy**: x is a part of y, e.g., WHEEL-BICYCLE; the inverse relationship is **holonymy**.

Classification of these relations can be performed by searching for characteristic patterns between pairs of words, e.g., *X, such as Y*, which signals hyponymy (Hearst, 1992), or *X but Y*, which signals antonymy (Hatzivassiloglou and McKeown, 1997). Another approach is to analyze each term's **distributional statistics** (the frequency of its neighboring words). Such approaches are described in detail in chapter 14.

4.2.2 Word sense disambiguation as classification

How can we tell living *plants* from manufacturing *plants*? The context is often critical:

(4.4) a. Town officials are hoping to attract new manufacturing plants through weakened environmental regulations.
b. The endangered plants play an important role in the local ecosystem.

It is possible to build a feature vector using the bag-of-words representation, by treating each context as a pseudo-document. The feature function is then,

$$f((\textit{plant}, \textit{The endangered plants play an} \ldots), y) = \\ \{(\textit{the}, y) : 1, (\textit{endangered}, y) : 1, (\textit{play}, y) : 1, (\textit{an}, y) : 1, \ldots\}$$

As in document classification, many of these features are irrelevant, but a few are very strong predictors. In this example, the context word *endangered* is a strong signal that the intended sense is biology rather than manufacturing. We would therefore expect a learning algorithm to assign high weight to $(\textit{endangered}, \text{BIOLOGY})$, and low weight to $(\textit{endangered}, \text{MANUFACTURING})$.[5]

It may also be helpful to go beyond the bag-of-words: for example, one might encode the position of each context word with respect to the target, e.g.,

$$f((\textit{bank}, \textit{I went to the bank to deposit my paycheck}), y) = \\ \{(i-3, \textit{went}, y) : 1, (i+2, \textit{deposit}, y) : 1, (i+4, \textit{paycheck}, y) : 1\}$$

These are called **collocation features**, and they give more information about the specific role played by each context word. This idea can be taken further by incorporating additional syntactic information about the grammatical role played by each context feature, such as the **dependency path** (see chapter 11).

[5]The context bag-of-words can be also used be used to perform word-sense disambiguation without machine learning: the Lesk (1986) algorithm selects the word sense whose dictionary definition best overlaps the local context.

Using such features, a classifier can be trained from labeled data. A **semantic concordance** is a corpus in which each open-class word (nouns, verbs, adjectives, and adverbs) is tagged with its word sense from the target dictionary or thesaurus. SemCor is a semantic concordance built from 234K tokens of the Brown corpus (Francis and Kucera, 1982), annotated as part of the WORDNET project (Fellbaum, 2010). SemCor annotations look like this:

(4.5) As of Sunday$^{1}_{N}$ night$^{1}_{N}$ there was$^{4}_{V}$ no word$^{2}_{N}$ ...,

with the superscripts indicating the annotated sense of each polysemous word, and the subscripts indicating the part-of-speech.

As always, supervised classification is only possible if enough labeled examples can be accumulated. This is difficult in word sense disambiguation, because each polysemous lemma requires its own training set: having a good classifier for the senses of *serve* is no help towards disambiguating *plant*. For this reason, unsupervised and **semi-supervised** methods are particularly important for word sense disambiguation (e.g., Yarowsky, 1995). These methods will be discussed in chapter 5. Unsupervised methods typically lean on the heuristic of "one sense per discourse", which means that a lemma will usually have a single, consistent sense throughout any given document (Gale et al., 1992). Based on this heuristic, we can propagate information from high-confidence instances to lower-confidence instances in the same document (Yarowsky, 1995). Semi-supervised methods combine labeled and unlabeled data, and are discussed in more detail in chapter 5.

4.3 Design decisions for text classification

Text classification involves a number of design decisions. In some cases, the design decision is clear from the mathematics: if you are using regularization, then a regularization weight λ must be chosen. Other decisions are more subtle, arising only in the low level "plumbing" code that ingests and processes the raw data. Such decision can be surprisingly consequential for classification accuracy.

4.3.1 What is a word?

The bag-of-words representation presupposes that extracting a vector of word counts from text is unambiguous. But text documents are generally represented as a sequences of characters (in an encoding such as ascii or unicode), and the conversion to bag-of-words presupposes a definition of the "words" that are to be counted.

Whitespace	Isn't	Ahab,	Ahab?	;)				
Treebank	Is	n't	Ahab	,	Ahab	?	;	)
Tweet	Isn't	Ahab	,	Ahab	?	;)		
TokTok (Dehdari, 2014)	Isn	'	t	Ahab	,	Ahab	?	;)

Figure 4.1: The output of four NLTK tokenizers, applied to the string *Isn't Ahab, Ahab? ;)*

Tokenization

The first subtask for constructing a bag-of-words vector is **tokenization**: converting the text from a sequence of characters to a sequence of **word!tokens**. A simple approach is to define a subset of characters as whitespace, and then split the text on these tokens. However, whitespace-based tokenization is not ideal: we may want to split conjunctions like *isn't* and hyphenated phrases like *prize-winning* and *half-asleep*, and we likely want to separate words from commas and periods that immediately follow them. At the same time, it would be better not to split abbreviations like *U.S.* and *Ph.D.* In languages with Roman scripts, tokenization is typically performed using regular expressions, with modules designed to handle each of these cases. For example, the NLTK package includes a number of tokenizers (Loper and Bird, 2002); the outputs of four of the better-known tokenizers are shown in Figure 4.1. Social media researchers have found that emoticons and other forms of orthographic variation pose new challenges for tokenization, leading to the development of special purpose tokenizers to handle these phenomena (O'Connor et al., 2010).

Tokenization is a language-specific problem, and each language poses unique challenges. For example, Chinese does not include spaces between words, nor any other consistent orthographic markers of word boundaries. A "greedy" approach is to scan the input for character substrings that are in a predefined lexicon. However, Xue et al. (2003) notes that this can be ambiguous, since many character sequences could be segmented in multiple ways. Instead, he trains a classifier to determine whether each Chinese character, or **hanzi**, is a word boundary. More advanced sequence labeling methods for word segmentation are discussed in § 8.4. Similar problems can occur in languages with alphabetic scripts, such as German, which does not include whitespace in compound nouns, yielding examples such as *Freundschaftsbezeigungen* (demonstration of friendship) and *Dilettantenaufdringlichkeiten* (the importunities of dilettantes). As Twain (1997) argues, "*These things are not words, they are alphabetic processions.*" Social media raises similar problems for English and other languages, with hashtags such as *#TrueLoveInFourWords* requiring decomposition for analysis (Brun and Roux, 2014).

Original	The	Williams	sisters	are	leaving	this	tennis	centre
Porter stemmer	the	william	sister	are	leav	thi	tenni	centr
Lancaster stemmer	the	william	sist	ar	leav	thi	ten	cent
WordNet lemmatizer	The	Williams	**sister**	are	leaving	this	tennis	centre

Figure 4.2: Sample outputs of the Porter (1980) and Lancaster (Paice, 1990) stemmers, and the WORDNET lemmatizer

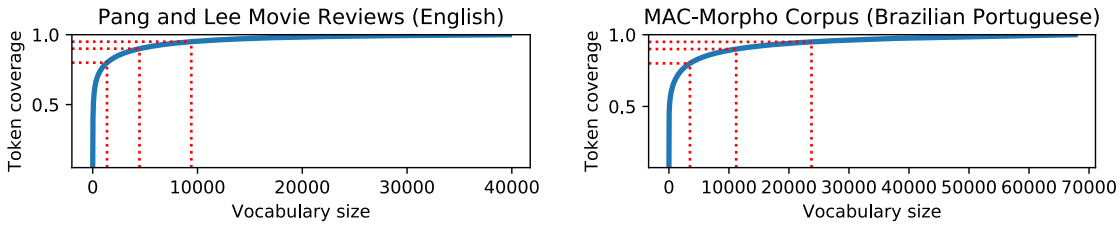
Text normalization

After splitting the text into tokens, the next question is which tokens are really distinct. Is it necessary to distinguish *great*, *Great*, and *GREAT*? Sentence-initial capitalization may be irrelevant to the classification task. Going further, the complete elimination of case distinctions will result in a smaller vocabulary, and thus smaller feature vectors. However, case distinctions might be relevant in some situations: for example, *apple* is a delicious pie filling, while *Apple* is a company that specializes in proprietary dongles and power adapters.

For Roman script, case conversion can be performed using unicode string libraries. Many scripts do not have case distinctions (e.g., the Devanagari script used for South Asian languages, the Thai alphabet, and Japanese kana), and case conversion for all scripts may not be available in every programming environment. (Unicode support is an important distinction between Python's versions 2 and 3, and is a good reason for migrating to Python 3 if you have not already done so. Compare the output of the code `"\à l\'hôtel".upper()` in the two language versions.)

Case conversion is a type of **text normalization**, which refers to string transformations that remove distinctions that are irrelevant to downstream applications (Sproat et al., 2001). Other forms of normalization include the standardization of numbers (e.g., *1,000* to *1000*) and dates (e.g., *August 11, 2015* to *2015/11/08*). Depending on the application, it may even be worthwhile to convert all numbers and dates to special tokens, `!NUM` and `!DATE`. In social media, there are additional orthographic phenomena that may be normalized, such as expressive lengthening, e.g., *cooooool* (Aw et al., 2006; Yang and Eisenstein, 2013). Similarly, historical texts feature spelling variations that may need to be normalized to a contemporary standard form (Baron and Rayson, 2008).

A more extreme form of normalization is to eliminate **inflectional affixes**, such as the *-ed* and *-s* suffixes in English. On this view, *whale*, *whales*, and *whaling* all refer to the same underlying concept, so they should be grouped into a single feature. A **stemmer** is a program for eliminating affixes, usually by applying a series of regular expression substitutions. Character-based stemming algorithms are necessarily approximate, as shown in Figure 4.2: the Lancaster stemmer incorrectly identifies *-ers* as an inflectional suffix of



(a) Movie review data in English

(b) News articles in Brazilian Portuguese

Figure 4.3: Tradeoff between token coverage (y-axis) and vocabulary size, on the NLTK movie review dataset, after sorting the vocabulary by decreasing frequency. The red dashed lines indicate 80%, 90%, and 95% coverage.

sisters (by analogy to *fix/fixers*), and both stemmers incorrectly identify *-s* as a suffix of *this* and *Williams*. Fortunately, even inaccurate stemming can improve bag-of-words classification models, by merging related strings and thereby reducing the vocabulary size.

Accurately handling irregular orthography requires word-specific rules. **Lemmatizers** are systems that identify the underlying lemma of a given wordform. They must avoid the over-generalization errors of the stemmers in Figure 4.2, and also handle more complex transformations, such as *geese→goose*. The output of the WordNet lemmatizer is shown in the final line of Figure 4.2. Both stemming and lemmatization are language-specific: an English stemmer or lemmatizer is of little use on a text written in another language. The discipline of **morphology** relates to the study of word-internal structure, and is described in more detail in § 9.1.2.

The value of normalization depends on the data and the task. Normalization reduces the size of the feature space, which can help in generalization. However, there is always the risk of merging away linguistically meaningful distinctions. In supervised machine learning, regularization and smoothing can play a similar role to normalization — preventing the learner from overfitting to rare features — while avoiding the language-specific engineering required for accurate normalization. In unsupervised scenarios, such as content-based information retrieval (Manning et al., 2008) and topic modeling (Blei et al., 2003), normalization is more critical.

4.3.2 How many words?

Limiting the size of the feature vector reduces the memory footprint of the resulting models, and increases the speed of prediction. Normalization can help to play this role, but a more direct approach is simply to limit the vocabulary to the N most frequent words in the dataset. For example, in the MOVIE-REVIEWS dataset provided with NLTK (originally from Pang et al., 2002), there are 39,768 word types, and 1.58M tokens. As shown

in Figure 4.3a, the most frequent 4000 word types cover 90% of all tokens, offering an order-of-magnitude reduction in the model size. Such ratios are language-specific: in for example, in the Brazilian Portuguese Mac-Morpho corpus (Aluísio et al., 2003), attaining 90% coverage requires more than 10000 word types (Figure 4.3b). This reflects the morphological complexity of Portuguese, which includes many more inflectional suffixes than English.

Eliminating rare words is not always advantageous for classification performance: for example, names, which are typically rare, play a large role in distinguishing topics of news articles. Another way to reduce the size of the feature space is to eliminate **stopwords** such as *the*, *to*, and *and*, which may seem to play little role in expressing the topic, sentiment, or stance. This is typically done by creating a **stoplist** (e.g., NLTK.CORPUS.STOPWORDS), and then ignoring all terms that match the list. However, corpus linguists and social psychologists have shown that seemingly inconsequential words can offer surprising insights about the author or nature of the text (Biber, 1991; Chung and Pennebaker, 2007). Furthermore, high-frequency words are unlikely to cause overfitting in discriminative classifiers. As with normalization, stopword filtering is more important for unsupervised problems, such as term-based document retrieval.

Another alternative for controlling model size is **feature hashing** (Weinberger et al., 2009). Each feature is assigned an index using a hash function. If a hash function that permits collisions is chosen (typically by taking the hash output modulo some integer), then the model can be made arbitrarily small, as multiple features share a single weight. Because most features are rare, accuracy is surprisingly robust to such collisions (Ganchev and Dredze, 2008).

4.3.3 Count or binary?

Finally, we may consider whether we want our feature vector to include the *count* of each word, or its *presence*. This gets at a subtle limitation of linear classification: it's worse to have two *failure*s than one, but is it really twice as bad? Motivated by this intuition, Pang et al. (2002) use binary indicators of presence or absence in the feature vector: $f_j(\boldsymbol{x}, y) \in \{0, 1\}$. They find that classifiers trained on these binary vectors tend to outperform feature vectors based on word counts. One explanation is that words tend to appear in clumps: if a word has appeared once in a document, it is likely to appear again (Church, 2000). These subsequent appearances can be attributed to this tendency towards repetition, and thus provide little additional information about the class label of the document.

4.4 Evaluating classifiers

In any supervised machine learning application, it is critical to reserve a held-out test set. This data should be used for only one purpose: to evaluate the overall accuracy of a single

classifier. Using this data more than once would cause the estimated accuracy to be overly optimistic, because the classifier would be customized to this data, and would not perform as well as on unseen data in the future. It is usually necessary to set hyperparameters or perform feature selection, so you may need to construct a **tuning** or **development set** for this purpose, as discussed in § 2.2.5.

There are a number of ways to evaluate classifier performance. The simplest is **accuracy**: the number of correct predictions, divided by the total number of instances,

$$\text{acc}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i}^{N} \delta(y^{(i)} = \hat{y}). \quad [4.4]$$

Exams are usually graded by accuracy. Why are other metrics necessary? The main reason is **class imbalance**. Suppose you are building a classifier to detect whether an electronic health record (EHR) describes symptoms of a rare disease, which appears in only 1% of all documents in the dataset. A classifier that reports $\hat{y} = \text{NEGATIVE}$ for all documents would achieve 99% accuracy, but would be practically useless. We need metrics that are capable of detecting the classifier's ability to discriminate between classes, even when the distribution is skewed.

One solution is to build a **balanced test set**, in which each possible label is equally represented. But in the EHR example, this would mean throwing away 98% of the original dataset! Furthermore, the detection threshold itself might be a design consideration: in health-related applications, we might prefer a very sensitive classifier, which returned a positive prediction if there is even a small chance that $y^{(i)} = \text{POSITIVE}$. In other applications, a positive result might trigger a costly action, so we would prefer a classifier that only makes positive predictions when absolutely certain. We need additional metrics to capture these characteristics.

4.4.1 Precision, recall, and F-MEASURE

For any label (e.g., positive for presence of symptoms of a disease), there are two possible errors:

- **False positive**: the system incorrectly predicts the label.
- **False negative**: the system incorrectly fails to predict the label.

Similarly, for any label, there are two ways to be correct:

- **True positive**: the system correctly predicts the label.
- **True negative**: the system correctly predicts that the label does not apply to this instance.

Classifiers that make a lot of false positives have low **precision**: they predict the label even when it isn't there. Classifiers that make a lot of false negatives have low **recall**: they fail to predict the label, even when it is there. These metrics distinguish these two sources of error, and are defined formally as:

$$\text{RECALL}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad [4.5]$$

$$\text{PRECISION}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) = \frac{\text{TP}}{\text{TP} + \text{FP}}. \quad [4.6]$$

Recall and precision are both conditional likelihoods of a correct prediction, which is why their numerators are the same. Recall is conditioned on k being the correct label, $y^{(i)} = k$, so the denominator sums over true positive and false negatives. Precision is conditioned on k being the prediction, so the denominator sums over true positives and false positives. Note that true negatives are not considered in either statistic. The classifier that labels every document as "negative" would achieve zero recall; precision would be $\frac{0}{0}$.

Recall and precision are complementary. A high-recall classifier is preferred when false positives are cheaper than false negatives: for example, in a preliminary screening for symptoms of a disease, the cost of a false positive might be an additional test, while a false negative would result in the disease going untreated. Conversely, a high-precision classifier is preferred when false positives are more expensive: for example, in spam detection, a false negative is a relatively minor inconvenience, while a false positive might mean that an important message goes unread.

The F**-MEASURE** combines recall and precision into a single metric, using the harmonic mean:

$$F\text{-MEASURE}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) = \frac{2rp}{r + p}, \quad [4.7]$$

where r is recall and p is precision.[6]

Evaluating multi-class classification Recall, precision, and F-MEASURE are defined with respect to a specific label k. When there are multiple labels of interest (e.g., in word sense disambiguation or emotion classification), it is necessary to combine the F-MEASURE across each class. **Macro** F**-MEASURE** is the average F-MEASURE across several classes,

$$\text{Macro-}F(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} F\text{-MEASURE}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) \quad [4.8]$$

[6] F-MEASURE is sometimes called F_1, and generalizes to $F_\beta = \frac{(1+\beta^2)rp}{\beta^2 p + r}$. The β parameter can be tuned to emphasize recall or precision.

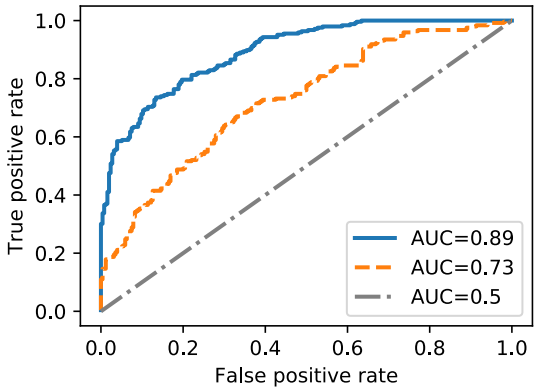


Figure 4.4: ROC curves for three classifiers of varying discriminative power, measured by AUC (area under the curve)

In multi-class problems with unbalanced class distributions, the macro F-MEASURE is a balanced measure of how well the classifier recognizes each class. In **micro F-MEASURE**, we compute true positives, false positives, and false negatives for each class, and then add them up to compute a single recall, precision, and F-MEASURE. This metric is balanced across instances rather than classes, so it weights each class in proportion to its frequency — unlike macro F-MEASURE, which weights each class equally.

4.4.2 Threshold-free metrics

In binary classification problems, it is possible to trade off between recall and precision by adding a constant "threshold" to the output of the scoring function. This makes it possible to trace out a curve, where each point indicates the performance at a single threshold. In the **receiver operating characteristic (ROC)** curve,[7] the x-axis indicates the **false positive rate**, $\frac{\text{FP}}{\text{FP+TN}}$, and the y-axis indicates the recall, or **true positive rate**. A perfect classifier attains perfect recall without any false positives, tracing a "curve" from the origin (0,0) to the upper left corner (0,1), and then to (1,1). In expectation, a non-discriminative classifier traces a diagonal line from the origin (0,0) to the upper right corner (1,1). Real classifiers tend to fall between these two extremes. Examples are shown in Figure 4.4.

The ROC curve can be summarized in a single number by taking its integral, the **area under the curve (AUC)**. The AUC can be interpreted as the probability that a randomly-selected positive example will be assigned a higher score by the classifier than a randomly-

[7] The name "receiver operator characteristic" comes from the metric's origin in signal processing applications (Peterson et al., 1954). Other threshold-free metrics include **precision-recall curves**, **precision-at-k**, and **balanced F-MEASURE**; see Manning et al. (2008) for more details.

selected negative example. A perfect classifier has $\text{AUC} = 1$ (all positive examples score higher than all negative examples); a non-discriminative classifier has $\text{AUC} = 0.5$ (given a randomly selected positive and negative example, either could score higher with equal probability); a perfectly wrong classifier would have $\text{AUC} = 0$ (all negative examples score higher than all positive examples). One advantage of AUC in comparison to F-MEASURE is that the baseline rate of 0.5 does not depend on the label distribution.

4.4.3 Classifier comparison and statistical significance

Natural language processing research and engineering often involves comparing different classification techniques. In some cases, the comparison is between algorithms, such as logistic regression versus averaged perceptron, or L_2 regularization versus L_1. In other cases, the comparison is between feature sets, such as the bag-of-words versus positional bag-of-words (see § 4.2.2). **Ablation testing** involves systematically removing (ablating) various aspects of the classifier, such as feature groups, and testing the **null hypothesis** that the ablated classifier is as good as the full model.

A full treatment of hypothesis testing is beyond the scope of this text, but this section contains a brief summary of the techniques necessary to compare classifiers. The main aim of hypothesis testing is to determine whether the difference between two statistics — for example, the accuracies of two classifiers — is likely to arise by chance. We will be concerned with chance fluctuations that arise due to the finite size of the test set.[8] An improvement of 10% on a test set with ten instances may reflect a random fluctuation that makes the test set more favorable to classifier c_1 than c_2; on another test set with a different ten instances, we might find that c_2 does better than c_1. But if we observe the same 10% improvement on a test set with 1000 instances, this is highly unlikely to be explained by chance. Such a finding is said to be **statistically significant** at a level p, which is the probability of observing an effect of equal or greater magnitude when the null hypothesis is true. The notation $p < .05$ indicates that the likelihood of an equal or greater effect is less than 5%, assuming the null hypothesis is true.[9]

The binomial test

The statistical significance of a difference in accuracy can be evaluated using classical tests, such as the **binomial test**.[10] Suppose that classifiers c_1 and c_2 disagree on N instances in a

[8] Other sources of variance include the initialization of non-convex classifiers such as neural networks, and the ordering of instances in online learning such as stochastic gradient descent and perceptron.

[9] Statistical hypothesis testing is useful only to the extent that the existing test set is representative of the instances that will be encountered in the future. If, for example, the test set is constructed from news documents, no hypothesis test can predict which classifier will perform best on documents from another domain, such as electronic health records.

[10] A well-known alternative to the binomial test is **McNemar's test**, which computes a **test statistic** based on the number of examples that are correctly classified by one system and incorrectly classified by the other.

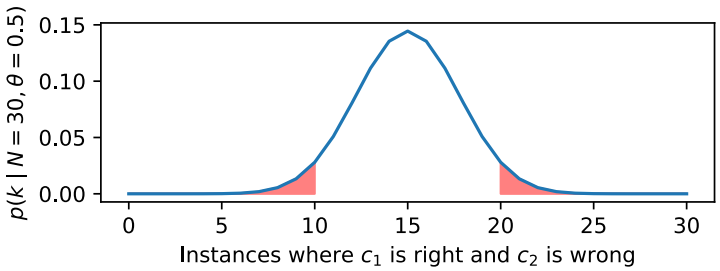


Figure 4.5: Probability mass function for the binomial distribution. The pink highlighted areas represent the cumulative probability for a significance test on an observation of $k = 10$ and $N = 30$.

test set with binary labels, and that c_1 is correct on k of those instances. Under the null hypothesis that the classifiers are equally accurate, we would expect k/N to be roughly equal to $1/2$, and as N increases, k/N should be increasingly close to this expected value. These properties are captured by the **binomial distribution**, which is a probability over counts of binary random variables. We write $k \sim \text{Binom}(\theta, N)$ to indicate that k is drawn from a binomial distribution, with parameter N indicating the number of random "draws", and θ indicating the probability of "success" on each draw. Each draw is an example on which the two classifiers disagree, and a "success" is a case in which c_1 is right and c_2 is wrong. (The label space is assumed to be binary, so if the classifiers disagree, exactly one of them is correct. The test can be generalized to multi-class classification by focusing on the examples in which exactly one classifier is correct.)

The **probability mass function** (PMF) of the binomial distribution is,

$$\text{p}_{\text{Binom}}(k; N, \theta) = \binom{N}{k} \theta^k (1-\theta)^{N-k}, \quad [4.9]$$

with θ^k representing the probability of the k successes, $(1-\theta)^{N-k}$ representing the probability of the $N-k$ unsuccessful draws. The expression $\binom{N}{k} = \frac{N!}{k!(N-k)!}$ is a binomial coefficient, representing the number of possible orderings of events; this ensures that the distribution sums to one over all $k \in \{0, 1, 2, \ldots, N\}$.

Under the null hypothesis, when the classifiers disagree, each classifier is equally likely to be right, so $\theta = \frac{1}{2}$. Now suppose that among N disagreements, c_1 is correct $k < \frac{N}{2}$ times. The probability of c_1 being correct k or fewer times is the **one-tailed p-value**,

The null hypothesis distribution for this test statistic is known to be drawn from a chi-squared distribution with a single degree of freedom, so a p-value can be computed from the cumulative density function of this distribution (Dietterich, 1998). Both tests give similar results in most circumstances, but the binomial test is easier to understand from first principles.

because it is computed from the area under the binomial probability mass function from 0 to k, as shown in the left tail of Figure 4.5. This **cumulative probability** is computed as a sum over all values $i \leq k$,

$$\Pr_{\text{Binom}}\left(\text{count}(\hat{y}_2^{(i)} = y^{(i)} \neq \hat{y}_1^{(i)}) \leq k; N, \theta = \frac{1}{2}\right) = \sum_{i=0}^{k} \text{p}_{\text{Binom}}\left(i; N, \theta = \frac{1}{2}\right). \quad [4.10]$$

The one-tailed p-value applies only to the asymmetric null hypothesis that c_1 is at least as accurate as c_2. To test the **two-tailed** null hypothesis that c_1 and c_2 are equally accurate, we would take the sum of one-tailed p-values, where the second term is computed from the right tail of Figure 4.5. The binomial distribution is symmetric, so this can be computed by simply doubling the one-tailed p-value.

Two-tailed tests are more stringent, but they are necessary in cases in which there is no prior intuition about whether c_1 or c_2 is better. For example, in comparing logistic regression versus averaged perceptron, a two-tailed test is appropriate. In an ablation test, c_2 may contain a superset of the features available to c_1. If the additional features are thought to be likely to improve performance, then a one-tailed test would be appropriate, if chosen in advance. However, such a test can only prove that c_2 is more accurate than c_1, and not the reverse.

***Randomized testing**

The binomial test is appropriate for accuracy, but not for more complex metrics such as F-MEASURE. To compute statistical significance for arbitrary metrics, we can apply randomization. Specifically, draw a set of M **bootstrap samples** (Efron and Tibshirani, 1993), by resampling instances from the original test set with replacement. Each bootstrap sample is itself a test set of size N. Some instances from the original test set will not appear in any given bootstrap sample, while others will appear multiple times; but overall, the sample will be drawn from the same distribution as the original test set. We can then compute any desired evaluation on each bootstrap sample, which gives a distribution over the value of the metric. Algorithm 7 shows how to perform this computation.

To compare the F-MEASURE of two classifiers c_1 and c_2, we set the function $\delta(\cdot)$ to compute the difference in F-MEASURE on the bootstrap sample. If the difference is less than or equal to zero in at least 5% of the samples, then we cannot reject the one-tailed null hypothesis that c_2 is at least as good as c_1 (Berg-Kirkpatrick et al., 2012). We may also be interested in the 95% **confidence interval** around a metric of interest, such as the F-MEASURE of a single classifier. This can be computed by sorting the output of Algorithm 7, and then setting the top and bottom of the 95% confidence interval to the values at the 2.5% and 97.5% percentiles of the sorted outputs. Alternatively, you can fit a normal distribution to the set of differences across bootstrap samples, and compute a Gaussian confidence interval from the mean and variance.

Algorithm 7 Bootstrap sampling for classifier evaluation. The original test set is $\{\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}\}$, the metric is $\delta(\cdot)$, and the number of samples is M.

procedure BOOTSTRAP-SAMPLE($\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}, \delta(\cdot), M$)
  for $t \in \{1, 2, \ldots, M\}$ **do**
    for $i \in \{1, 2, \ldots, N\}$ **do**
      $j \sim \text{UniformInteger}(1, N)$
      $\tilde{\boldsymbol{x}}^{(i)} \leftarrow \boldsymbol{x}^{(j)}$
      $\tilde{\boldsymbol{y}}^{(i)} \leftarrow \boldsymbol{y}^{(j)}$
    $d^{(t)} \leftarrow \delta(\tilde{\boldsymbol{x}}^{(1:N)}, \tilde{\boldsymbol{y}}^{(1:N)})$
  return $\{d^{(t)}\}_{t=1}^{M}$

As the number of bootstrap samples goes to infinity, $M \rightarrow \infty$, the bootstrap estimate is increasingly accurate. A typical choice for M is 10^4 or 10^5; larger numbers of samples are necessary for smaller p-values. One way to validate your choice of M is to run the test multiple times, and ensure that the p-values are similar; if not, increase M by an order of magnitude. This is a heuristic measure of the **variance** of the test, which can decreases with the square root $\sqrt{M}$ (Robert and Casella, 2013).

4.4.4 *Multiple comparisons

Sometimes it is necessary to perform multiple hypothesis tests, such as when comparing the performance of several classifiers on multiple datasets. Suppose you have five datasets, and you compare four versions of your classifier against a baseline system, for a total of 20 comparisons. Even if none of your classifiers is better than the baseline, there will be some chance variation in the results, and in expectation you will get one statistically significant improvement at $p = 0.05 = \frac{1}{20}$. It is therefore necessary to adjust the p-values when reporting the results of multiple comparisons.

One approach is to require a threshold of $\frac{\alpha}{m}$ to report a p value of $p < \alpha$ when performing m tests. This is known as the **Bonferroni correction**, and it limits the overall probability of incorrectly rejecting the null hypothesis at α. Another approach is to bound the **false discovery rate** (FDR), which is the fraction of null hypothesis rejections that are incorrect. Benjamini and Hochberg (1995) propose a p-value correction that bounds the fraction of false discoveries at α: sort the p-values of each individual test in ascending order, and set the significance threshold equal to largest k such that $p_k \leq \frac{k}{m}\alpha$. If $k > 1$, the FDR adjustment is more permissive than the Bonferroni correction.

4.5 Building datasets

Sometimes, if you want to build a classifier, you must first build a dataset of your own. This includes selecting a set of documents or instances to annotate, and then performing the annotations. The scope of the dataset may be determined by the application: if you want to build a system to classify electronic health records, then you must work with a corpus of records of the type that your classifier will encounter when deployed. In other cases, the goal is to build a system that will work across a broad range of documents. In this case, it is best to have a *balanced* corpus, with contributions from many styles and genres. For example, the Brown corpus draws from texts ranging from government documents to romance novels (Francis, 1964), and the Google Web Treebank includes annotations for five "domains" of web documents: question answers, emails, newsgroups, reviews, and blogs (Petrov and McDonald, 2012).

4.5.1 Metadata as labels

Annotation is difficult and time-consuming, and most people would rather avoid it. It is sometimes possible to exploit existing metadata to obtain labels for training a classifier. For example, reviews are often accompanied by a numerical rating, which can be converted into a classification label (see § 4.1). Similarly, the nationalities of social media users can be estimated from their profiles (Dredze et al., 2013) or even the time zones of their posts (Gouws et al., 2011). More ambitiously, we may try to classify the political affiliations of social media profiles based on their social network connections to politicians and major political parties (Rao et al., 2010).

The convenience of quickly constructing large labeled datasets without manual annotation is appealing. However this approach relies on the assumption that unlabeled instances — for which metadata is unavailable — will be similar to labeled instances. Consider the example of labeling the political affiliation of social media users based on their network ties to politicians. If a classifier attains high accuracy on such a test set, is it safe to assume that it accurately predicts the political affiliation of all social media users? Probably not. Social media users who establish social network ties to politicians may be more likely to mention politics in the text of their messages, as compared to the average user, for whom no political metadata is available. If so, the accuracy on a test set constructed from social network metadata would give an overly optimistic picture of the method's true performance on unlabeled data.

4.5.2 Labeling data

In many cases, there is no way to get ground truth labels other than manual annotation. An annotation protocol should satisfy several criteria: the annotations should be *expressive* enough to capture the phenomenon of interest; they should be *replicable*, meaning that

another annotator or team of annotators would produce very similar annotations if given the same data; and they should be *scalable*, so that they can be produced relatively quickly. Hovy and Lavid (2010) propose a structured procedure for obtaining annotations that meet these criteria, which is summarized below.

1. **Determine what to annotate**. This is usually based on some theory of the underlying phenomenon: for example, if the goal is to produce annotations about the emotional state of a document's author, one should start with a theoretical account of the types or dimensions of emotion (e.g., Mohammad and Turney, 2013). At this stage, the tradeoff between expressiveness and scalability should be considered: a full instantiation of the underlying theory might be too costly to annotate at scale, so reasonable approximations should be considered.

2. Optionally, one may **design or select a software tool to support the annotation effort**. Existing general-purpose annotation tools include BRAT (Stenetorp et al., 2012) and MMAX2 (Müller and Strube, 2006).

3. **Formalize the instructions for the annotation task**. To the extent that the instructions are not explicit, the resulting annotations will depend on the intuitions of the annotators. These intuitions may not be shared by other annotators, or by the users of the annotated data. Therefore explicit instructions are critical to ensuring the annotations are replicable and usable by other researchers.

4. **Perform a pilot annotation** of a small subset of data, with multiple annotators for each instance. This will give a preliminary assessment of both the replicability and scalability of the current annotation instructions. Metrics for computing the rate of agreement are described below. Manual analysis of specific disagreements should help to clarify the instructions, and may lead to modifications of the annotation task itself. For example, if two labels are commonly conflated by annotators, it may be best to merge them.

5. **Annotate the data**. After finalizing the annotation protocol and instructions, the main annotation effort can begin. Some, if not all, of the instances should receive multiple annotations, so that inter-annotator agreement can be computed. In some annotation projects, instances receive many annotations, which are then aggregated into a "consensus" label (e.g., Danescu-Niculescu-Mizil et al., 2013). However, if the annotations are time-consuming or require significant expertise, it may be preferable to maximize scalability by obtaining multiple annotations for only a small subset of examples.

6. **Compute and report inter-annotator agreement, and release the data**. In some cases, the raw text data cannot be released, due to concerns related to copyright or

privacy. In these cases, one solution is to publicly release **stand-off annotations**, which contain links to document identifiers. The documents themselves can be released under the terms of a licensing agreement, which can impose conditions on how the data is used. It is important to think through the potential consequences of releasing data: people may make personal data publicly available without realizing that it could be redistributed in a dataset and publicized far beyond their expectations (boyd and Crawford, 2012).

Measuring inter-annotator agreement

To measure the replicability of annotations, a standard practice is to compute the extent to which annotators agree with each other. If the annotators frequently disagree, this casts doubt on either their reliability or on the annotation system itself. For classification, one can compute the frequency with which the annotators agree; for rating scales, one can compute the average distance between ratings. These raw agreement statistics must then be compared with the rate of agreement by chance — the expected level of agreement that would be obtained between two annotators who ignored the data.

Cohen's Kappa is widely used for quantifying the agreement on discrete labeling tasks (Cohen, 1960; Carletta, 1996),[11]

$$\kappa = \frac{\text{agreement} - E[\text{agreement}]}{1 - E[\text{agreement}]}. \quad [4.11]$$

The numerator is the difference between the observed agreement and the chance agreement, and the denominator is the difference between perfect agreement and chance agreement. Thus, $\kappa = 1$ when the annotators agree in every case, and $\kappa = 0$ when the annotators agree only as often as would happen by chance. Various heuristic scales have been proposed for determining when κ indicates "moderate", "good", or "substantial" agreement; for reference, Lee and Narayanan (2005) report $\kappa \approx 0.45 - 0.47$ for annotations of emotions in spoken dialogues, which they describe as "moderate agreement"; Stolcke et al. (2000) report $\kappa = 0.8$ for annotations of **dialogue acts**, which are labels for the purpose of each turn in a conversation.

When there are two annotators, the expected chance agreement is computed as,

$$E[\text{agreement}] = \sum_k \hat{\Pr}(Y = k)^2, \quad [4.12]$$

where k is a sum over labels, and $\hat{\Pr}(Y = k)$ is the empirical probability of label k across all annotations. The formula is derived from the expected number of agreements if the annotations were randomly shuffled. Thus, in a binary labeling task, if one label is applied to 90% of instances, chance agreement is $.9^2 + .1^2 = .82$.

[11] For other types of annotations, Krippendorf's alpha is a popular choice (Hayes and Krippendorff, 2007; Artstein and Poesio, 2008).

Crowdsourcing

Crowdsourcing is often used to rapidly obtain annotations for classification problems. For example, **Amazon Mechanical Turk** makes it possible to define "human intelligence tasks (hits)", such as labeling data. The researcher sets a price for each set of annotations and a list of minimal qualifications for annotators, such as their native language and their satisfaction rate on previous tasks. The use of relatively untrained "crowdworkers" contrasts with earlier annotation efforts, which relied on professional linguists (Marcus et al., 1993). However, crowdsourcing has been found to produce reliable annotations for many language-related tasks (Snow et al., 2008). Crowdsourcing is part of the broader field of **human computation** (Law and Ahn, 2011).For a critical examination of ethical issues related to crowdsourcing, see Fort et al. (2011).

Additional resources

Many of the preprocessing issues discussed in this chapter also arise in information retrieval. See Manning et al. (2008) for discussion of tokenization and related algorithms. For more on hypothesis testing in particular and replicability in general, see (Dror et al., 2017, 2018).

Exercises

1. As noted in § 4.3.3, words tend to appear in clumps, with subsequent occurrences of a word being more probable. More concretely, if word j has probability $\phi_{y,j}$ of appearing in a document with label y, then the probability of two appearances $(x_j^{(i)} = 2)$ is greater than $\phi_{y,j}^2$.

   Suppose you are applying Naïve Bayes to a binary classification. Focus on a word j which is more probable under label $y = 1$, so that,

   $$\Pr(w = j \mid y = 1) > \Pr(w = j \mid y = 0). \quad [4.13]$$

   Now suppose that $x_j^{(i)} > 1$. All else equal, will the classifier overestimate or underestimate the posterior $\Pr(y = 1 \mid \boldsymbol{x})$?

2. Prove that F-measure is never greater than the arithmetic mean of recall and precision, $\frac{r+p}{2}$. Your solution should also show that F-measure is equal to $\frac{r+p}{2}$ iff $r = p$.

3. Given a binary classification problem in which the probability of the "positive" label is equal to α, what is the expected F-MEASURE of a random classifier which ignores the data, and selects $\hat{y} = +1$ with probability $\frac{1}{2}$? (Assume that $\mathrm{p}(\hat{y}) \perp \mathrm{p}(y)$.) What is the expected F-MEASURE of a classifier that selects $\hat{y} = +1$ with probability α (also independent of $y^{(i)}$)? Depending on α, which random classifier will score better?

4. Suppose that binary classifiers c_1 and c_2 disagree on $N = 30$ cases, and that c_1 is correct in $k = 10$ of those cases.

   - Write a program that uses primitive functions such as $\exp$ and factorial to compute the **two-tailed** p-value — you may use an implementation of the "choose" function if one is avaiable. Verify your code against the output of a library for computing the binomial test or the binomial CDF, such as SCIPY.STATS.BINOM in Python.
   - Then use a randomized test to try to obtain the same p-value. In each sample, draw from a binomial distribution with $N = 30$ and $\theta = \frac{1}{2}$. Count the fraction of samples in which $k \leq 10$. This is the one-tailed p-value; double this to compute the two-tailed p-value.
   - Try this with varying numbers of bootstrap samples: $M \in \{100, 1000, 5000, 10000\}$. For $M = 100$ and $M = 1000$, run the test 10 times, and plot the resulting p-values.
   - Finally, perform the same tests for $N = 70$ and $k = 25$.

5. SemCor 3.0 is a labeled dataset for word sense disambiguation. You can download it,[12] or access it in NLTK.CORPORA.SEMCOR.

   Choose a word that appears at least ten times in SemCor (*find*), and annotate its WordNet senses across ten randomly-selected examples, without looking at the ground truth. Use online WordNet to understand the definition of each of the senses.[13] Have a partner do the same annotations, and compute the raw rate of agreement, expected chance rate of agreement, and Cohen's kappa.

6. Download the Pang and Lee movie review data, currently available from `http://www.cs.cornell.edu/people/pabo/movie-review-data/`. Hold out a randomly-selected 400 reviews as a test set.

   Download a sentiment lexicon, such as the one currently available from Bing Liu, `https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html`. Tokenize the data, and classify each document as positive iff it has more positive sentiment words than negative sentiment words. Compute the accuracy and F-MEASURE on detecting positive reviews on the test set, using this lexicon-based classifier.

   Then train a discriminative classifier (averaged perceptron or logistic regression) on the training set, and compute its accuracy and F-MEASURE on the test set.

   Determine whether the differences are statistically significant, using two-tailed hypothesis tests: Binomial for the difference in accuracy, and bootstrap for the difference in macro-F-MEASURE.

[12] e.g., `https://github.com/google-research-datasets/word_sense_disambigation_corpora` or `http://globalwordnet.org/wordnet-annotated-corpora/`

[13] http://wordnetweb.princeton.edu/perl/webwn

The remaining problems will require you to build a classifier and test its properties. Pick a multi-class text classification dataset that is not already tokenized. One example is a dataset of New York Times headlines and topics (Boydstun, 2013).[14] Divide your data into training (60%), development (20%), and test sets (20%), if no such division already exists. If your dataset is very large, you may want to focus on a few thousand instances at first.

7. Compare various vocabulary sizes of $10^2, 10^3, 10^4, 10^5$, using the most frequent words in each case (you may use any reasonable tokenizer). Train logistic regression classifiers for each vocabulary size, and apply them to the development set. Plot the accuracy and Macro-F-MEASURE with the increasing vocabulary size. For each vocabulary size, tune the regularizer to maximize accuracy on a subset of data that is held out from the training set.

8. Compare the following tokenization algorithms:

   - Whitespace, using a regular expression;
   - The Penn Treebank tokenizer from NLTK;
   - Splitting the input into non-overlapping five-character units, regardless of whitespace or punctuation.

   Compute the token/type ratio for each tokenizer on the training data, and explain what you find. Train your classifier on each tokenized dataset, tuning the regularizer on a subset of data that is held out from the training data. Tokenize the development set, and report accuracy and Macro-F-MEASURE.

9. Apply the Porter and Lancaster stemmers to the training set, using any reasonable tokenizer, and compute the token/type ratios. Train your classifier on the stemmed data, and compute the accuracy and Macro-F-MEASURE on stemmed development data, again using a held-out portion of the training data to tune the regularizer.

10. Identify the best combination of vocabulary filtering, tokenization, and stemming from the previous three problems. Apply this preprocessing to the test set, and compute the test set accuracy and Macro-F-MEASURE. Compare against a baseline system that applies no vocabulary filtering, whitespace tokenization, and no stemming.

    Use the binomial test to determine whether your best-performing system is significantly more accurate than the baseline.

[14] Available as a CSV file at `http://www.amber-boydstun.com/supplementary-information-for-making-the-news.html`. Use the field TOPIC_2DIGIT for this problem.

Use the bootstrap test with $M = 10^4$ to determine whether your best-performing system achieves significantly higher macro-F-MEASURE.

Chapter 5

Learning without supervision

So far, we have assumed the following setup:

- a **training set** where you get observations $\boldsymbol{x}$ and labels y;
- a **test set** where you only get observations $\boldsymbol{x}$.

Without labeled data, is it possible to learn anything? This scenario is known as **unsupervised learning**, and we will see that indeed it is possible to learn about the underlying structure of unlabeled observations. This chapter will also explore some related scenarios: **semi-supervised learning**, in which only some instances are labeled, and **domain adaptation**, in which the training data differs from the data on which the trained system will be deployed.

5.1 Unsupervised learning

To motivate unsupervised learning, consider the problem of word sense disambiguation (§ 4.2). The goal is to classify each instance of a word, such as *bank* into a sense,

- `bank#1`: a financial institution
- `bank#2`: the land bordering a river

It is difficult to obtain sufficient training data for word sense disambiguation, because even a large corpus will contain only a few instances of all but the most common words. Is it possible to learn anything about these different senses without labeled data?

Word sense disambiguation is usually performed using feature vectors constructed from the local context of the word to be disambiguated. For example, for the word

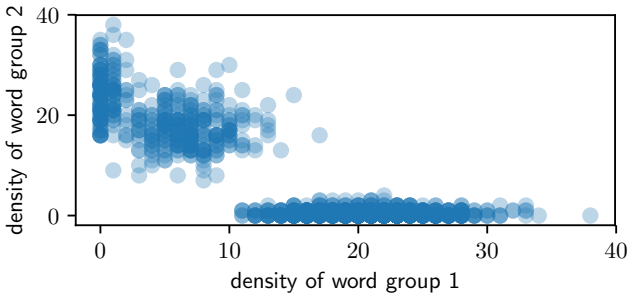


Figure 5.1: Counts of words from two different context groups

bank, the immediate context might typically include words from one of the following two groups:

1. *financial, deposits, credit, lending, capital, markets, regulated, reserve, liquid, assets*
2. *land, water, geography, stream, river, flow, deposits, discharge, channel, ecology*

Now consider a scatterplot, in which each point is a document containing the word *bank*. The location of the document on the x-axis is the count of words in group 1, and the location on the y-axis is the count for group 2. In such a plot, shown in Figure 5.1, two "blobs" might emerge, and these blobs correspond to the different senses of *bank*.

Here's a related scenario, from a different problem. Suppose you download thousands of news articles, and make a scatterplot, where each point corresponds to a document: the x-axis is the frequency of the group of words (*hurricane, winds, storm*); the y-axis is the frequency of the group (*election, voters, vote*). This time, three blobs might emerge: one for documents that are largely about a hurricane, another for documents largely about a election, and a third for documents about neither topic.

These clumps represent the underlying structure of the data. But the two-dimensional scatter plots are based on groupings of context words, and in real scenarios these word lists are unknown. Unsupervised learning applies the same basic idea, but in a high-dimensional space with one dimension for every context word. This space can't be directly visualized, but the goal is the same: try to identify the underlying structure of the observed data, such that there are a few clusters of points, each of which is internally coherent. **Clustering** algorithms are capable of finding such structure automatically.

5.1.1 K-means clustering

Clustering algorithms assign each data point to a discrete cluster, $z_i \in 1, 2, \ldots K$. One of the best known clustering algorithms is **K-means**, an iterative algorithm that maintains

Algorithm 8 K-means clustering algorithm

procedure K-MEANS($\boldsymbol{x}_{1:N}, K$)
  for $i \in 1 \dots N$ **do** ▷ initialize cluster memberships
    $z^{(i)} \leftarrow \text{RANDOMINT}(1, K)$
  repeat
    for $k \in 1 \dots K$ **do** ▷ recompute cluster centers
      $\boldsymbol{\nu}_k \leftarrow \frac{1}{\delta(z^{(i)}=k)} \sum_{i=1}^{N} \delta(z^{(i)} = k)\boldsymbol{x}^{(i)}$
    for $i \in 1 \dots N$ **do** ▷ reassign instances to nearest clusters
      $z^{(i)} \leftarrow \text{argmin}_k \, ||\boldsymbol{x}^{(i)} - \boldsymbol{\nu}_k||^2$
  until converged
  return $\{z^{(i)}\}$ ▷ return cluster assignments

a cluster assignment for each instance, and a central ("mean") location for each cluster. K-means iterates between updates to the assignments and the centers:

1. each instance is placed in the cluster with the closest center;

2. each center is recomputed as the average over points in the cluster.

This procedure is formalized in Algorithm 8. The term $||\boldsymbol{x}^{(i)} - \boldsymbol{\nu}||^2$ refers to the squared Euclidean norm, $\sum_{j=1}^{V}(x_j^{(i)} - \nu_j)^2$. An important property of K-means is that the converged solution depends on the initialization, and a better clustering can sometimes be found simply by re-running the algorithm from a different random starting point.

Soft K-means is a particularly relevant variant. Instead of directly assigning each point to a specific cluster, soft K-means assigns to each point a *distribution* over clusters $\boldsymbol{q}^{(i)}$, so that $\sum_{k=1}^{K} q^{(i)}(k) = 1$, and $\forall_k, q^{(i)}(k) \geq 0$. The soft weight $q^{(i)}(k)$ is computed from the distance of $\boldsymbol{x}^{(i)}$ to the cluster center $\boldsymbol{\nu}_k$. In turn, the center of each cluster is computed from a weighted average of the points in the cluster,

$$\boldsymbol{\nu}_k = \frac{1}{\sum_{i=1}^{N} q^{(i)}(k)} \sum_{i=1}^{N} q^{(i)}(k)\boldsymbol{x}^{(i)}. \quad [5.1]$$

We will now explore a probablistic version of soft K-means clustering, based on **expectation-maximization** (EM). Because EM clustering can be derived as an approximation to maximum-likelihood estimation, it can be extended in a number of useful ways.

5.1.2 Expectation-Maximization (EM)

Expectation-maximization combines the idea of soft K-means with Naïve Bayes classification. To review, Naïve Bayes defines a probability distribution over the data,

$$\log \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\phi}, \boldsymbol{\mu}) = \sum_{i=1}^{N} \log \Big(\mathrm{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}; \boldsymbol{\phi}) \times \mathrm{p}(y^{(i)}; \boldsymbol{\mu}) \Big) \tag{5.2}$$

Now suppose that you never observe the labels. To indicate this, we'll refer to the label of each instance as $z^{(i)}$, rather than $y^{(i)}$, which is usually reserved for observed variables. By marginalizing over the **latent variables** $\boldsymbol{z}$, we obtain the marginal probability of the observed instances $\boldsymbol{x}$:

$$\log \mathrm{p}(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{\mu}) = \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu}) \tag{5.3}$$

$$= \sum_{i=1}^{N} \log \sum_{z=1}^{K} \mathrm{p}(\boldsymbol{x}^{(i)}, z; \boldsymbol{\phi}, \boldsymbol{\mu}) \tag{5.4}$$

$$= \sum_{i=1}^{N} \log \sum_{z=1}^{K} \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \times \mathrm{p}(z; \boldsymbol{\mu}). \tag{5.5}$$

The parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$ can be obtained by maximizing the marginal likelihood in Equation 5.5. Why is this the right thing to maximize? Without labels, discriminative learning is impossible — there's nothing to discriminate. So maximum likelihood is all we have.

When the labels are observed, we can estimate the parameters of the Naïve Bayes probability model separately for each label. But marginalizing over the labels couples these parameters, making direct optimization of $\log \mathrm{p}(\boldsymbol{x})$ intractable. We will approximate the log-likelihood by introducing an auxiliary variable $\boldsymbol{q}^{(i)}$, which is a distribution over the label set $\mathcal{Z} = \{1, 2, \ldots, K\}$. The optimization procedure will alternate between updates to $\boldsymbol{q}$ and updates to the parameters $(\boldsymbol{\phi}, \boldsymbol{\mu})$. Thus, $\boldsymbol{q}^{(i)}$ plays here as in soft K-means.

To derive the updates for this optimization, multiply the right side of Equation 5.5 by

the ratio $\frac{q^{(i)}(z)}{q^{(i)}(z)} = 1$,

$$\log \mathrm{p}(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{\mu}) = \sum_{i=1}^{N} \log \sum_{z=1}^{K} \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \times \mathrm{p}(z; \boldsymbol{\mu}) \times \frac{q^{(i)}(z)}{q^{(i)}(z)} \tag{5.6}$$

$$= \sum_{i=1}^{N} \log \sum_{z=1}^{K} q^{(i)}(z) \times \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \times \mathrm{p}(z; \boldsymbol{\mu}) \times \frac{1}{q^{(i)}(z)} \tag{5.7}$$

$$= \sum_{i=1}^{N} \log E_{\boldsymbol{q}^{(i)}} \left[\frac{\mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \mathrm{p}(z; \boldsymbol{\mu})}{q^{(i)}(z)} \right], \tag{5.8}$$

where $E_{\boldsymbol{q}^{(i)}}[f(z)] = \sum_{z=1}^{K} q^{(i)}(z) \times f(z)$ refers to the expectation of the function f under the distribution $z \sim \boldsymbol{q}^{(i)}$.

Jensen's inequality says that because $\log$ is a concave function, we can push it inside the expectation, and obtain a lower bound.

$$\log \mathrm{p}(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{\mu}) \geq \sum_{i=1}^{N} E_{\boldsymbol{q}^{(i)}} \left[\log \frac{\mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \mathrm{p}(z; \boldsymbol{\mu})}{q^{(i)}(z)} \right] \tag{5.9}$$

$$J \triangleq \sum_{i=1}^{N} E_{\boldsymbol{q}^{(i)}} \left[\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \boldsymbol{\mu}) - \log q^{(i)}(z) \right] \tag{5.10}$$

$$= \sum_{i=1}^{N} E_{\boldsymbol{q}^{(i)}} \left[\log \mathrm{p}(\boldsymbol{x}^{(i)}, z; \boldsymbol{\phi}, \boldsymbol{\mu}) \right] + H(\boldsymbol{q}^{(i)}) \tag{5.11}$$

We will focus on Equation 5.10, which is the lower bound on the marginal log-likelihood of the observed data, $\log \mathrm{p}(\boldsymbol{x})$. Equation 5.11 shows the connection to the information theoretic concept of **entropy**, $H(\boldsymbol{q}^{(i)}) = -\sum_{z=1}^{K} q^{(i)}(z) \log q^{(i)}(z)$, which measures the average amount of information produced by a draw from the distribution $q^{(i)}$. The lower bound J is a function of two groups of arguments:

- the distributions $\boldsymbol{q}^{(i)}$ for each instance;
- the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$.

The expectation-maximization (EM) algorithm maximizes the bound with respect to each of these arguments in turn, while holding the other fixed.

The E-step

The step in which we update $\boldsymbol{q}^{(i)}$ is known as the **E-step**, because it updates the distribution under which the expectation is computed. To derive this update, first write out the

expectation in the lower bound as a sum,

$$J = \sum_{i=1}^{N} \sum_{z=1}^{K} q^{(i)}(z) \left[\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \boldsymbol{\mu}) - \log q^{(i)}(z) \right]. \tag{5.12}$$

When optimizing this bound, we must also respect a set of "sum-to-one" constraints, $\sum_{z=1}^{K} q^{(i)}(z) = 1$ for all i. Just as in Naïve Bayes, this constraint can be incorporated into a Lagrangian:

$$J_q = \sum_{i=1}^{N} \sum_{z=1}^{K} q^{(i)}(z) \left(\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \mu) - \log q^{(i)}(z) \right) + \lambda^{(i)}(1 - \sum_{z=1}^{K} q^{(i)}(z)), \tag{5.13}$$

where $\lambda^{(i)}$ is the Lagrange multiplier for instance i.

The Lagrangian is maximized by taking the derivative and solving for $q^{(i)}$:

$$\frac{\partial J_q}{\partial q^{(i)}(z)} = \log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \boldsymbol{\theta}) - \log q^{(i)}(z) - 1 - \lambda^{(i)} \tag{5.14}$$

$$\log q^{(i)}(z) = \log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \mu) - 1 - \lambda^{(i)} \tag{5.15}$$

$$q^{(i)}(z) \propto \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \times \mathrm{p}(z; \mu). \tag{5.16}$$

Applying the sum-to-one constraint gives an exact solution,

$$q^{(i)}(z) = \frac{\mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \times \mathrm{p}(z; \boldsymbol{\mu})}{\sum_{z'=1}^{K} \mathrm{p}(\boldsymbol{x}^{(i)} \mid z'; \boldsymbol{\phi}) \times \mathrm{p}(z'; \boldsymbol{\mu})} \tag{5.17}$$

$$= \mathrm{p}(z \mid \boldsymbol{x}^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu}). \tag{5.18}$$

After normalizing, each $\boldsymbol{q}^{(i)}$ — which is the soft distribution over clusters for data $\boldsymbol{x}^{(i)}$ — is set to the posterior probability $\mathrm{p}(z \mid \boldsymbol{x}^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu})$ under the current parameters. Although the Lagrange multipliers $\lambda^{(i)}$ were introduced as additional parameters, they drop out during normalization.

The M-step

Next, we hold fixed the soft assignments $\boldsymbol{q}^{(i)}$, and **m**aximize with respect to the parameters, $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$. Let's focus on the parameter $\boldsymbol{\phi}$, which parametrizes the likelihood $\mathrm{p}(\boldsymbol{x} \mid z; \boldsymbol{\phi})$, and leave $\boldsymbol{\mu}$ for an exercise. The parameter $\boldsymbol{\phi}$ is a distribution over words for each cluster, so it is optimized under the constraint that $\sum_{j=1}^{V} \phi_{z,j} = 1$. To incorporate this

constraint, we introduce a set of Lagrange multiplers $\{\lambda_z\}_{z=1}^K$, and from the Lagrangian,

$$J_\phi = \sum_{i=1}^{N}\sum_{z=1}^{K} q^{(i)}(z)\left(\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z;\boldsymbol{\phi}) + \log \mathrm{p}(z;\mu) - \log q^{(i)}(z)\right) + \sum_{z=1}^{K}\lambda_z(1-\sum_{j=1}^{V}\phi_{z,j}). \quad [5.19]$$

The term $\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z;\boldsymbol{\phi})$ is the conditional log-likelihood for the multinomial, which expands to,

$$\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z,\boldsymbol{\phi}) = C + \sum_{j=1}^{V} x_j \log \phi_{z,j}, \quad [5.20]$$

where C is a constant with respect to $\boldsymbol{\phi}$ — see Equation 2.12 in § 2.2 for more discussion of this probability function.

Setting the derivative of J_ϕ equal to zero,

$$\frac{\partial J_\phi}{\partial \phi_{z,j}} = \sum_{i=1}^{N} q^{(i)}(z) \times \frac{x_j^{(i)}}{\phi_{z,j}} - \lambda_z \quad [5.21]$$

$$\phi_{z,j} \propto \sum_{i=1}^{N} q^{(i)}(z) \times x_j^{(i)}. \quad [5.22]$$

Because $\boldsymbol{\phi}_z$ is constrained to be a probability distribution, the exact solution is computed as,

$$\phi_{z,j} = \frac{\sum_{i=1}^{N} q^{(i)}(z) \times x_j^{(i)}}{\sum_{j'=1}^{V}\sum_{i=1}^{N} q^{(i)}(z) \times x_{j'}^{(i)}} = \frac{E_q\left[\text{count}(z,j)\right]}{\sum_{j'=1}^{V} E_q\left[\text{count}(z,j')\right]}, \quad [5.23]$$

where the counter $j \in \{1,2,\ldots,V\}$ indexes over base features, such as words.

This update sets $\boldsymbol{\phi}_z$ equal to the relative frequency estimate of the *expected counts* under the distribution $\boldsymbol{q}$. As in supervised Naïve Bayes, we can smooth these counts by adding a constant α. The update for $\boldsymbol{\mu}$ is similar: $\mu_z \propto \sum_{i=1}^{N} q^{(i)}(z) = E_q\left[\text{count}(z)\right]$, which is the expected frequency of cluster z. These probabilities can also be smoothed. In sum, the M-step is just like Naïve Bayes, but with expected counts rather than observed counts.

The multinomial likelihood $\mathrm{p}(\boldsymbol{x} \mid z)$ can be replaced with other probability distributions: for example, for continuous observations, a Gaussian distribution can be used. In some cases, there is no closed-form update to the parameters of the likelihood. One approach is to run gradient-based optimization at each M-step; another is to simply take a single step along the gradient step and then return to the E-step (Berg-Kirkpatrick et al., 2010).

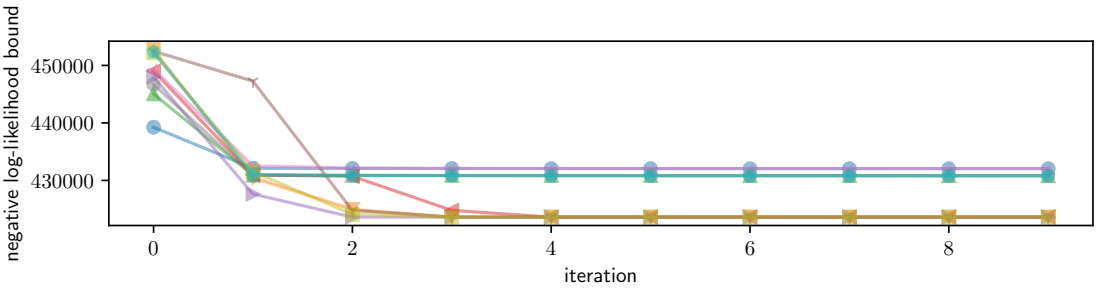


Figure 5.2: Sensitivity of expectation-maximization to initialization. Each line shows the progress of optimization from a different random initialization.

5.1.3 EM as an optimization algorithm

Algorithms that update a global objective by alternating between updates to subsets of the parameters are called **coordinate ascent** algorithms. The objective J (the lower bound on the marginal likelihood of the data) is separately convex in $\boldsymbol{q}$ and $(\boldsymbol{\mu}, \boldsymbol{\phi})$, but it is not jointly convex in all terms; this condition is known as **biconvexity**. Each step of the expectation-maximization algorithm is guaranteed not to decrease the lower bound J, which means that EM will converge towards a solution at which no nearby points yield further improvements. This solution is a **local optimum** — it is as good or better than any of its immediate neighbors, but is *not* guaranteed to be optimal among all possible configurations of $(\boldsymbol{q}, \boldsymbol{\mu}, \boldsymbol{\phi})$.

The fact that there is no guarantee of global optimality means that initialization is important: where you start can determine where you finish. To illustrate this point, Figure 5.2 shows the objective function for EM with ten different random initializations: while the objective function improves monotonically in each run, it converges to several different values.[1] For the convex objectives that we encountered in chapter 2, it was not necessary to worry about initialization, because gradient-based optimization guaranteed to reach the global minimum. But in expectation-maximization — as in the deep neural networks from chapter 3 — initialization matters.

In **hard EM**, each $\boldsymbol{q}^{(i)}$ distribution assigns probability of 1 to a single label $\hat{z}^{(i)}$, and zero probability to all others (Neal and Hinton, 1998). This is similar in spirit to K-means clustering, and can outperform standard EM in some cases (Spitkovsky et al., 2010). Another variant of expectation-maximization incorporates stochastic gradient descent (SGD): after performing a local E-step at each instance $\boldsymbol{x}^{(i)}$, we immediately make a gradient update to the parameters $(\boldsymbol{\mu}, \boldsymbol{\phi})$. This algorithm has been called **incremental expectation maximization** (Neal and Hinton, 1998) and **online expectation maximization** (Sato and Ishii,

[1] The figure shows the upper bound on the *negative* log-likelihood, because optimization is typically framed as minimization rather than maximization.

2000; Cappé and Moulines, 2009), and is especially useful when there is no closed-form optimum for the likelihood $\text{p}(\boldsymbol{x} \mid z)$, and in online settings where new data is constantly streamed in (see Liang and Klein, 2009, for a comparison for online EM variants).

5.1.4 How many clusters?

So far, we have assumed that the number of clusters K is given. In some cases, this assumption is valid. For example, a lexical semantic resource like WORDNET might define the number of senses for a word. In other cases, the number of clusters could be a parameter for the user to tune: some readers want a coarse-grained clustering of news stories into three or four clusters, while others want a fine-grained clustering into twenty or more. But many times there is little extrinsic guidance for how to choose K.

One solution is to choose the number of clusters to maximize a metric of clustering quality. The other parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$ are chosen to maximize the log-likelihood bound J, so this might seem a potential candidate for tuning K. However, J will never decrease with K: if it is possible to obtain a bound of J_K with K clusters, then it is always possible to do at least as well with $K + 1$ clusters, by simply ignoring the additional cluster and setting its probability to zero in $\boldsymbol{q}$ and $\boldsymbol{\mu}$. It is therefore necessary to introduce a penalty for model complexity, so that fewer clusters are preferred. For example, the Akaike Information Crition (AIC; Akaike, 1974) is the linear combination of the number of parameters and the log-likelihood,

$$\text{AIC} = 2M - 2J, \tag{5.24}$$

where M is the number of parameters. In an expectation-maximization clustering algorithm, $M = K \times V + K$. Since the number of parameters increases with the number of clusters K, the AIC may prefer more parsimonious models, even if they do not fit the data quite as well.

Another choice is to maximize the **predictive likelihood** on heldout data. This data is not used to estimate the model parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$, and so it is not the case that the likelihood on this data is guaranteed to increase with K. Figure 5.3 shows the negative log-likelihood on training and heldout data, as well as the AIC.

***Bayesian nonparametrics** An alternative approach is to treat the number of clusters as another latent variable. This requires statistical inference over a set of models with a variable number of clusters. This is not possible within the framework of expectation-maximization, but there are several alternative inference procedures which can be applied, including **Markov Chain Monte Carlo (MCMC)**, which is briefly discussed in § 5.5 (for more details, see Chapter 25 of Murphy, 2012). Bayesian nonparametrics have been applied to the problem of unsupervised word sense induction, learning not only the word senses but also the number of senses per word (Reisinger and Mooney, 2010).

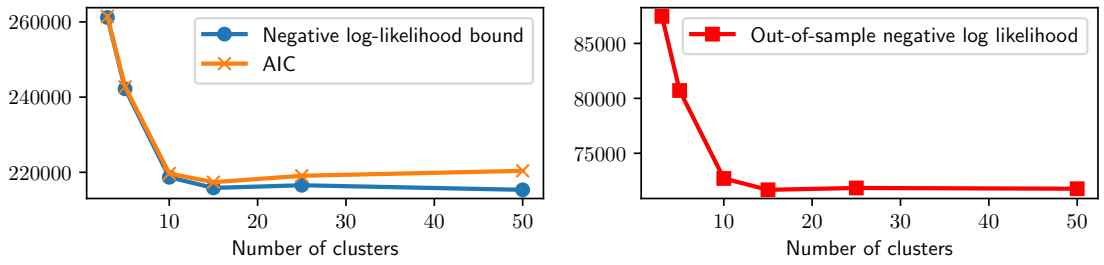


Figure 5.3: The negative log-likelihood and AIC for several runs of expectation-maximization, on synthetic data. Although the data was generated from a model with $K = 10$, the optimal number of clusters is $\hat{K} = 15$, according to AIC and the heldout log-likelihood. The training set log-likelihood continues to improve as K increases.

5.2 Applications of expectation-maximization

EM is not really an "algorithm" like, say, quicksort. Rather, it is a framework for learning with missing data. The recipe for using EM on a problem of interest is:

- Introduce latent variables $\boldsymbol{z}$, such that it is easy to write the probability $P(\boldsymbol{x}, \boldsymbol{z})$. It should also be easy to estimate the associated parameters, given knowledge of $\boldsymbol{z}$.
- Derive the E-step updates for $q(\boldsymbol{z})$, which is typically factored as $q(\boldsymbol{z}) = \prod_{i=1}^{N} q_{z^{(i)}}(z^{(i)})$, where i is an index over instances.
- The M-step updates typically correspond to the soft version of a probabilistic supervised learning algorithm, like Naïve Bayes.

This section discusses a few of the many applications of this general framework.

5.2.1 Word sense induction

The chapter began by considering the problem of word sense disambiguation when the senses are not known in advance. Expectation-maximization can be applied to this problem by treating each cluster as a word sense. Each instance represents the use of an ambiguous word, and $\boldsymbol{x}^{(i)}$ is a vector of counts for the other words that appear nearby: Schütze (1998) uses all words within a 50-word window. The probability $\text{p}(\boldsymbol{x}^{(i)} \mid z)$ can be set to the multinomial distribution, as in Naïve Bayes. The EM algorithm can be applied directly to this data, yielding clusters that (hopefully) correspond to the word senses.

Better performance can be obtained by first applying **singular value decomposition** (SVD) to the matrix of context-counts $\mathbf{C}_{ij} = \text{count}(i, j)$, where $\text{count}(i, j)$ is the count of word j in the context of instance i. **Truncated** singular value decomposition approximates

the matrix $\mathbf{C}$ as a product of three matrices, $\mathbf{U}, \mathbf{S}, \mathbf{V}$, under the constraint that $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{S}$ is diagonal:

$$\begin{aligned} &\min_{\mathbf{U},\mathbf{S},\mathbf{V}} ||\mathbf{C} - \mathbf{U}\mathbf{S}\mathbf{V}^\top||_F \\ &\quad s.t. \mathbf{U} \in \mathbb{R}^{V \times K}, \mathbf{U}\mathbf{U}^\top = \mathbb{I} \\ &\quad\quad \mathbf{S} = \text{Diag}(s_1, s_2, \ldots, s_K) \\ &\quad\quad \mathbf{V}^\top \in \mathbb{R}^{N_p \times K}, \mathbf{V}\mathbf{V}^\top = \mathbb{I}, \end{aligned} \tag{5.25}$$

where $||\cdot||_F$ is the **Frobenius norm**, $||X||_F = \sqrt{\sum_{i,j} X_{i,j}^2}$. The matrix $\mathbf{U}$ contains the left singular vectors of $\mathbf{C}$, and the rows of this matrix can be used as low-dimensional representations of the count vectors $\boldsymbol{c}_i$. EM clustering can be made more robust by setting the instance descriptions $\boldsymbol{x}^{(i)}$ equal to these rows, rather than using raw counts (Schütze, 1998). However, because the instances are now dense vectors of continuous numbers, the probability $\text{p}(\boldsymbol{x}^{(i)} \mid z)$ must be defined as a multivariate Gaussian distribution.

In truncated singular value decomposition, the hyperparameter K is the truncation limit: when K is equal to the rank of $\mathbf{C}$, the norm of the difference between the original matrix $\mathbf{C}$ and its reconstruction $\mathbf{U}\mathbf{S}\mathbf{V}^\top$ will be zero. Lower values of K increase the reconstruction error, but yield vector representations that are smaller and easier to learn from. Singular value decomposition is discussed in more detail in chapter 14.

5.2.2 Semi-supervised learning

Expectation-maximization can also be applied to the problem of **semi-supervised learning**: learning from both labeled and unlabeled data in a single model. Semi-supervised learning makes use of annotated examples, ensuring that each label y corresponds to the desired concept. By adding unlabeled examples, it is possible to cover a greater fraction of the features than would appear in labeled data alone. Other methods for semi-supervised learning are discussed in § 5.3, but for now, let's approach the problem within the framework of expectation-maximization (Nigam et al., 2000).

Suppose we have labeled data $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N_\ell}$, and unlabeled data $\{\boldsymbol{x}^{(i)}\}_{i=N_\ell+1}^{N_\ell+N_u}$, where N_ℓ is the number of labeled instances and N_u is the number of unlabeled instances. We can learn from the combined data by maximizing a lower bound on the joint log-likelihood,

$$\mathcal{L} = \sum_{i=1}^{N_\ell} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\mu}, \boldsymbol{\phi}) + \sum_{j=N_\ell+1}^{N_\ell+N_u} \log \text{p}(\boldsymbol{x}^{(j)}; \boldsymbol{\mu}, \boldsymbol{\phi}) \tag{5.26}$$

$$= \sum_{i=1}^{N_\ell} \Big(\log \text{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}; \boldsymbol{\phi}) + \log \text{p}(y^{(i)}; \boldsymbol{\mu}) \Big) + \sum_{j=N_\ell+1}^{N_\ell+N_u} \log \sum_{y=1}^{K} \text{p}(\boldsymbol{x}^{(j)}, y; \boldsymbol{\mu}, \boldsymbol{\phi}). \tag{5.27}$$

Algorithm 9 Generative process for the Naïve Bayes classifier with hidden components

for Instance $i \in \{1, 2, \ldots, N\}$ **do**:
  Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$;
  Draw the component $z^{(i)} \sim \text{Categorical}(\boldsymbol{\beta}_{y^{(i)}})$;
  Draw the word counts $\boldsymbol{x}^{(i)} \mid y^{(i)}, z^{(i)} \sim \text{Multinomial}(\boldsymbol{\phi}_{z^{(i)}})$.

The left sum is identical to the objective in Naïve Bayes; the right sum is the marginal log-likelihood for expectation-maximization clustering, from Equation 5.5. We can construct a lower bound on this log-likelihood by introducing distributions $\boldsymbol{q}^{(j)}$ for all $j \in \{N_\ell + 1, \ldots, N_\ell + N_u\}$. The E-step updates these distributions; the M-step updates the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$, using the expected counts from the unlabeled data and the observed counts from the labeled data.

A critical issue in semi-supervised learning is how to balance the impact of the labeled and unlabeled data on the classifier weights, especially when the unlabeled data is much larger than the labeled dataset. The risk is that the unlabeled data will dominate, causing the parameters to drift towards a "natural clustering" of the instances — which may not correspond to a good classifier for the labeled data. One solution is to heuristically reweight the two components of Equation 5.26, tuning the weight of the two components on a heldout development set (Nigam et al., 2000).

5.2.3 Multi-component modeling

As a final application, let's return to fully supervised classification. A classic dataset for text classification is 20 newsgroups, which contains posts to a set of online forums, called newsgroups. One of the newsgroups is `comp.sys.mac.hardware`, which discusses Apple computing hardware. Suppose that within this newsgroup there are two kinds of posts: reviews of new hardware, and question-answer posts about hardware problems. The language in these *components* of the `mac.hardware class` might have little in common; if so, it would be better to model these components separately, rather than treating their union as a single class. However, the component responsible for each instance is not directly observed.

Recall that Naïve Bayes is based on a generative process, which provides a stochastic explanation for the observed data. In Naïve Bayes, each label is drawn from a categorical distribution with parameter $\boldsymbol{\mu}$, and each vector of word counts is drawn from a multinomial distribution with parameter $\boldsymbol{\phi}_y$. For multi-component modeling, we envision a slightly different generative process, incorporating both the observed label $y^{(i)}$ and the latent component $z^{(i)}$. This generative process is shown in Algorithm 9. A new parameter $\boldsymbol{\beta}_{y^{(i)}}$ defines the distribution of components, conditioned on the label $y^{(i)}$. The component, and not the class label, then parametrizes the distribution over words.

(5.1)	☺	Villeneuve a bel et bien **réussi** son pari de changer de perspectives tout en assurant une cohérence à la franchise.[2]
(5.2)	☹	Il est également trop **long** et bancal dans sa narration, tiède dans ses intentions, et tiraillé entre deux personnages et directions qui ne parviennent pas à coexister en harmonie.[3]
(5.3)		Denis Villeneuve a **réussi** une suite **parfaitement** maitrisée[4]
(5.4)		**Long**, **bavard**, hyper design, à peine agité (le comble de l'action : une bagarre dans la flotte), métaphysique et, surtout, ennuyeux jusqu'à la catalepsie.[5]
(5.5)		Une suite d'une écrasante puissance, mêlant **parfaitement** le contemplatif au narratif.[6]
(5.6)		Le film impitoyablement **bavard** finit quand même par se taire quand se lève l'espèce de bouquet final où semble se déchaîner, comme en libre parcours de poulets décapités, l'armée des graphistes numériques griffant nerveusement la palette graphique entre agonie et orgasme.[7]

Table 5.1: Labeled and unlabeled reviews of the films *Blade Runner 2049* and *Transformers: The Last Knight*.

The labeled data includes $(\boldsymbol{x}^{(i)}, y^{(i)})$, but not $z^{(i)}$, so this is another case of missing data. Again, we sum over the missing data, applying Jensen's inequality to as to obtain a lower bound on the log-likelihood,

$$\log \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}) = \log \sum_{z=1}^{K_z} \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}, z; \boldsymbol{\mu}, \boldsymbol{\phi}, \boldsymbol{\beta}) \tag{5.28}$$

$$\geq \log \mathrm{p}(y^{(i)}; \boldsymbol{\mu}) + E_{q_{Z|Y}^{(i)}}[\log \mathrm{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z \mid y^{(i)}; \boldsymbol{\beta}) - \log q^{(i)}(z)]. \tag{5.29}$$

We are now ready to apply expectation-maximization. As usual, the E-step updates the distribution over the missing data, $q_{Z|Y}^{(i)}$. The M-step updates the parameters,

$$\beta_{y,z} = \frac{E_q\left[\text{count}(y, z)\right]}{\sum_{z'=1}^{K_z} E_q\left[\text{count}(y, z')\right]} \tag{5.30}$$

$$\phi_{z,j} = \frac{E_q\left[\text{count}(z, j)\right]}{\sum_{j'=1}^{V} E_q\left[\text{count}(z, j')\right]}. \tag{5.31}$$

5.3 Semi-supervised learning

In semi-supervised learning, the learner makes use of both labeled and unlabeled data. To see how this could help, suppose you want to do sentiment analysis in French. In Ta-

ble 5.1, there are two labeled examples, one positive and one negative. From this data, a learner could conclude that *réussi* is positive and *long* is negative. This isn't much! However, we can propagate this information to the unlabeled data, and potentially learn more.

- If we are confident that *réussi* is positive, then we might guess that (5.3) is also positive.
- That suggests that *parfaitement* is also positive.
- We can then propagate this information to (5.5), and learn from the words in this example.
- Similarly, we can propagate from the labeled data to (5.4), which we guess to be negative because it shares the word *long*. This suggests that *bavard* is also negative, which we propagate to (5.6).

Instances (5.3) and (5.4) were "similar" to the labeled examples for positivity and negativity, respectively. By using these instances to expand the models for each class, it became possible to correctly label instances (5.5) and (5.6), which didn't share any important features with the original labeled data. This requires a key assumption: that similar instances will have similar labels.

In § 5.2.2, we discussed how expectation-maximization can be applied to semi-supervised learning. Using the labeled data, the initial parameters ϕ would assign a high weight for *réussi* in the positive class, and a high weight for *long* in the negative class. These weights helped to shape the distributions $\boldsymbol{q}$ for instances (5.3) and (5.4) in the E-step. In the next iteration of the M-step, the parameters ϕ are updated with counts from these instances, making it possible to correctly label the instances (5.5) and (5.6).

However, expectation-maximization has an important disadvantage: it requires using a generative classification model, which restricts the features that can be used for classification. In this section, we explore non-probabilistic approaches, which impose fewer restrictions on the classification model.

5.3.1 Multi-view learning

EM semi-supervised learning can be viewed as **self-training**: the labeled data guides the initial estimates of the classification parameters; these parameters are used to compute a label distribution over the unlabeled instances, $\boldsymbol{q}^{(i)}$; the label distributions are used to update the parameters. The risk is that self-training drifts away from the original labeled data. This problem can be ameliorated by **multi-view learning**. Here we take the assumption that the features can be decomposed into multiple "views", each of which is conditionally independent, given the label. For example, consider the problem of classifying a name as a person or location: one view is the name itself; another is the context in which it appears. This situation is illustrated in Table 5.2.

	$\boldsymbol{x}^{(1)}$	$\boldsymbol{x}^{(2)}$	y
1.	Peachtree Street	located on	LOC
2.	Dr. Walker	said	PER
3.	Zanzibar	located in	? $\rightarrow$ LOC
4.	Zanzibar	flew to	? $\rightarrow$ LOC
5.	Dr. Robert	recommended	? $\rightarrow$ PER
6.	Oprah	recommended	? $\rightarrow$ PER

Table 5.2: Example of multiview learning for named entity classification

Co-training is an iterative multi-view learning algorithm, in which there are separate classifiers for each view (Blum and Mitchell, 1998). At each iteration of the algorithm, each classifier predicts labels for a subset of the unlabeled instances, using only the features available in its view. These predictions are then used as ground truth to train the classifiers associated with the other views. In the example shown in Table 5.2, the classifier on $\boldsymbol{x}^{(1)}$ might correctly label instance #5 as a person, because of the feature *Dr*; this instance would then serve as training data for the classifier on $\boldsymbol{x}^{(2)}$, which would then be able to correctly label instance #6, thanks to the feature *recommended*. If the views are truly independent, this procedure is robust to drift. Furthermore, it imposes no restrictions on the classifiers that can be used for each view.

Word-sense disambiguation is particularly suited to multi-view learning, thanks to the heuristic of "one sense per discourse": if a polysemous word is used more than once in a given text or conversation, all usages refer to the same sense (Gale et al., 1992). This motivates a multi-view learning approach, in which one view corresponds to the local context (the surrounding words), and another view corresponds to the global context at the document level (Yarowsky, 1995). The local context view is first trained on a small seed dataset. We then identify its most confident predictions on unlabeled instances. The global context view is then used to extend these confident predictions to other instances within the same documents. These new instances are added to the training data to the local context classifier, which is retrained and then applied to the remaining unlabeled data.

5.3.2 Graph-based algorithms

Another family of approaches to semi-supervised learning begins by constructing a graph, in which pairs of instances are linked with symmetric weights $\omega_{i,j}$, e.g.,

$$\omega_{i,j} = \exp(-\alpha \times ||\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}||^2). \quad [5.32]$$

The goal is to use this weighted graph to propagate labels from a small set of labeled instances to larger set of unlabeled instances.

In **label propagation**, this is done through a series of matrix operations (Zhu et al., 2003). Let $\mathbf{Q}$ be a matrix of size $N \times K$, in which each row $\boldsymbol{q}^{(i)}$ describes the labeling of instance i. When ground truth labels are available, then $\boldsymbol{q}^{(i)}$ is an indicator vector, with $q^{(i)}_{y^{(i)}} = 1$ and $q^{(i)}_{y' \neq y^{(i)}} = 0$. Let us refer to the submatrix of rows containing labeled instances as $\mathbf{Q}_L$, and the remaining rows as $\mathbf{Q}_U$. The rows of $\mathbf{Q}_U$ are initialized to assign equal probabilities to all labels, $q_{i,k} = \frac{1}{K}$.

Now, let $T_{i,j}$ represent the "transition" probability of moving from node j to node i,

$$T_{i,j} \triangleq \Pr(j \rightarrow i) = \frac{\omega_{i,j}}{\sum_{k=1}^{N} \omega_{k,j}}. \quad [5.33]$$

We compute values of $T_{i,j}$ for all instances j and all *unlabeled* instances i, forming a matrix of size $N_U \times N$. If the dataset is large, this matrix may be expensive to store and manipulate; a solution is to sparsify it, by keeping only the κ largest values in each row, and setting all other values to zero. We can then "propagate" the label distributions to the unlabeled instances,

$$\tilde{\mathbf{Q}}_U \leftarrow \mathbf{T}\mathbf{Q} \quad [5.34]$$

$$\boldsymbol{s} \leftarrow \tilde{\mathbf{Q}}_U \mathbf{1} \quad [5.35]$$

$$\mathbf{Q}_U \leftarrow \text{Diag}(\boldsymbol{s})^{-1} \tilde{\mathbf{Q}}_U. \quad [5.36]$$

The expression $\tilde{\mathbf{Q}}_U \mathbf{1}$ indicates multiplication of $\tilde{\mathbf{Q}}_U$ by a column vector of ones, which is equivalent to computing the sum of each row of $\tilde{\mathbf{Q}}_U$. The matrix $\text{Diag}(\boldsymbol{s})$ is a diagonal matrix with the elements of $\boldsymbol{s}$ on the diagonals. The product $\text{Diag}(\boldsymbol{s})^{-1}\tilde{\mathbf{Q}}_U$ has the effect of normalizing the rows of $\tilde{\mathbf{Q}}_U$, so that each row of $\mathbf{Q}_U$ is a probability distribution over labels.

5.4 Domain adaptation

In many practical scenarios, the labeled data differs in some key respect from the data to which the trained model is to be applied. A classic example is in consumer reviews: we may have labeled reviews of movies (the source domain), but we want to predict the reviews of appliances (the target domain). A similar issue arises with genre differences: most linguistically-annotated data is news text, but application domains range from social media to electronic health records. In general, there may be several source and target domains, each with their own properties; however, for simplicity, this discussion will focus mainly on the case of a single source and target domain.

The simplest approach is "direct transfer": train a classifier on the source domain, and apply it directly to the target domain. The accuracy of this approach depends on the extent to which features are shared across domains. In review text, words like *outstanding* and

disappointing will apply across both movies and appliances; but others, like *terrifying*, may have meanings that are domain-specific. As a result, direct transfer performs poorly: for example, an out-of-domain classifier (trained on book reviews) suffers twice the error rate of an in-domain classifier on reviews of kitchen appliances (Blitzer et al., 2007). **Domain adaptation** algorithms attempt to do better than direct transfer by learning from data in both domains. There are two main families of domain adaptation algorithms, depending on whether any labeled data is available in the target domain.

5.4.1 Supervised domain adaptation

In supervised domain adaptation, there is a small amount of labeled data in the target domain, and a large amount of data in the source domain. The simplest approach would be to ignore domain differences, and simply merge the training data from the source and target domains. There are several other baseline approaches to dealing with this scenario (Daumé III, 2007):

Interpolation. Train a classifier for each domain, and combine their predictions, e.g.,

$$\hat{y} = \operatorname*{argmax}_y \lambda_s \Psi_s(\boldsymbol{x}, y) + (1 - \lambda_s)\Psi_t(\boldsymbol{x}, y), \quad [5.37]$$

where Ψ_s and Ψ_t are the scoring functions from the source and target domain classifiers respectively, and λ_s is the interpolation weight.

Prediction. Train a classifier on the source domain data, use its prediction as an additional feature in a classifier trained on the target domain data,

$$\hat{y}_s = \operatorname*{argmax}_y \Psi_s(\boldsymbol{x}, y) \quad [5.38]$$

$$\hat{y}_t = \operatorname*{argmax}_y \Psi_t([\boldsymbol{x}; \hat{y}_S], y). \quad [5.39]$$

Priors. Train a classifier on the source domain data, and use its weights as a prior distribution on the weights of the classifier for the target domain data. This is equivalent to regularizing the target domain weights towards the weights of the source domain classifier (Chelba and Acero, 2006),

$$\ell(\boldsymbol{\theta}_t) = \sum_{i=1}^{N} \ell^{(i)}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}_t) + \lambda ||\boldsymbol{\theta}_t - \boldsymbol{\theta}_s||_2^2, \quad [5.40]$$

where $\ell^{(i)}$ is the prediction loss on instance i, and λ is the regularization weight.

An effective and "frustratingly simple" alternative is EASYADAPT (Daumé III, 2007), which creates copies of each feature: one for each domain and one for the cross-domain setting. For example, a negative review of the film *Wonder Woman* begins, *As boring and flavorless as a three-day-old grilled cheese sandwich....*[8] The resulting bag-of-words feature vector would be,

$$\begin{aligned}\boldsymbol{f}(\boldsymbol{x}, y, d) = \{&(\textit{boring}, \frownie, \text{MOVIE}) : 1, (\textit{boring}, \frownie, *) : 1, \\ &(\textit{flavorless}, \frownie, \text{MOVIE}) : 1, (\textit{flavorless}, \frownie, *) : 1, \\ &(\textit{three-day-old}, \frownie, \text{MOVIE}) : 1, (\textit{three-day-old}, \frownie, *) : 1, \\ &\ldots\},\end{aligned}$$

with $(\textit{boring}, \frownie, \text{MOVIE})$ indicating the word *boring* appearing in a negative labeled document in the MOVIE domain, and $(\textit{boring}, \frownie, *)$ indicating the same word in a negative labeled document in *any* domain. It is up to the learner to allocate weight between the domain-specific and cross-domain features: for words that facilitate prediction in both domains, the learner will use the cross-domain features; for words that are relevant only to a single domain, the domain-specific features will be used. Any discriminative classifier can be used with these augmented features.[9]

5.4.2 Unsupervised domain adaptation

In unsupervised domain adaptation, there is no labeled data in the target domain. Unsupervised domain adaptation algorithms cope with this problem by trying to make the data from the source and target domains as similar as possible. This is typically done by learning a **projection function**, which puts the source and target data in a shared space, in which a learner can generalize across domains. This projection is learned from data in both domains, and is applied to the base features — for example, the bag-of-words in text classification. The projected features can then be used both for training and for prediction.

Linear projection

In linear projection, the cross-domain representation is constructed by a matrix-vector product,

$$\boldsymbol{g}(\boldsymbol{x}^{(i)}) = \mathbf{U}\boldsymbol{x}^{(i)}. \quad [5.41]$$

The projected vectors $\boldsymbol{g}(\boldsymbol{x}^{(i)})$ can then be used as base features during both training (from the source domain) and prediction (on the target domain).

[8] `http://www.colesmithey.com/capsules/2017/06/wonder-woman.HTML`, accessed October 9. 2017.

[9] EASYADAPT can be explained as a hierarchical Bayesian model, in which the weights for each domain are drawn from a shared prior (Finkel and Manning, 2009).

The projection matrix $\mathbf{U}$ can be learned in a number of different ways, but many approaches focus on compressing and reconstructing the base features (Ando and Zhang, 2005). For example, we can define a set of **pivot features**, which are typically chosen because they appear in both domains: in the case of review documents, pivot features might include evaluative adjectives like *outstanding* and *disappointing* (Blitzer et al., 2007). For each pivot feature j, we define an auxiliary problem of predicting whether the feature is present in each example, using the remaining base features. Let ϕ_j denote the weights of this classifier, and us horizontally concatenate the weights for each of the N_p pivot features into a matrix $\mathbf{\Phi} = [\phi_1, \phi_2, \ldots, \phi_{N_P}]$.

We then perform truncated singular value decomposition on $\mathbf{\Phi}$, as described in § 5.2.1, obtaining $\mathbf{\Phi} \approx \mathbf{U}\mathbf{S}\mathbf{V}^\top$. The rows of the matrix $\mathbf{U}$ summarize information about each base feature: indeed, the truncated singular value decomposition identifies a low-dimension basis for the weight matrix Φ, which in turn links base features to pivot features. Suppose that a base feature *reliable* occurs only in the target domain of appliance reviews. Nonetheless, it will have a positive weight towards some pivot features (e.g., *outstanding*, *recommended*), and a negative weight towards others (e.g., *worthless*, *unpleasant*). A base feature such as *watchable* might have the same associations with the pivot features, and therefore, $\boldsymbol{u}_{\text{reliable}} \approx \boldsymbol{u}_{\text{watchable}}$. The matrix $\mathbf{U}$ can thus project the base features into a space in which this information is shared.

Non-linear projection

Non-linear transformations of the base features can be accomplished by implementing the transformation function as a deep neural network, which is trained from an auxiliary objective.

Denoising objectives One possibility is to train a projection function to reconstruct a corrupted version of the original input. The original input can be corrupted in various ways: by the addition of random noise (Glorot et al., 2011; Chen et al., 2012), or by the deletion of features (Chen et al., 2012; Yang and Eisenstein, 2015). Denoising objectives share many properties of the linear projection method described above: they enable the projection function to be trained on large amounts of unlabeled data from the target domain, and allow information to be shared across the feature space, thereby reducing sensitivity to rare and domain-specific features.

Adversarial objectives The ultimate goal is for the transformed representations $\boldsymbol{g}(\boldsymbol{x}^{(i)})$ to be domain-general. This can be made an explicit optimization criterion by computing the similarity of transformed instances both within and between domains (Tzeng et al., 2015), or by formulating an auxiliary classification task, in which the domain itself is treated as a label (Ganin et al., 2016). This setting is **adversarial**, because we want

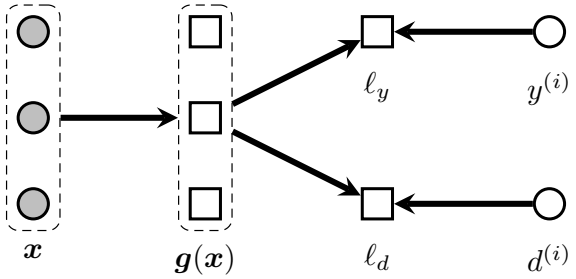


Figure 5.4: A schematic view of adversarial domain adaptation. The loss ℓ_y is computed only for instances from the source domain, where labels $y^{(i)}$ are available.

to learn a representation that makes this classifier perform poorly. At the same time, we want $\boldsymbol{g}(\boldsymbol{x}^{(i)})$ to enable accurate predictions of the labels $y^{(i)}$.

To formalize this idea, let $d^{(i)}$ represent the domain of instance i, and let $\ell_d(\boldsymbol{g}(\boldsymbol{x}^{(i)}), d^{(i)}; \boldsymbol{\theta}_d)$ represent the loss of a classifier (typically a deep neural network) trained to predict $d^{(i)}$ from the transformed representation $\boldsymbol{g}(\boldsymbol{x}^{(i)})$, using parameters $\boldsymbol{\theta}_d$. Analogously, let $\ell_y(\boldsymbol{g}(\boldsymbol{x}^{(i)}), y^{(i)}; \boldsymbol{\theta}_y)$ represent the loss of a classifier trained to predict the label $y^{(i)}$ from $\boldsymbol{g}(\boldsymbol{x}^{(i)})$, using parameters $\boldsymbol{\theta}_y$. The transformation $\boldsymbol{g}$ can then be trained from two criteria: it should yield accurate predictions of the labels $y^{(i)}$, while making *inaccurate* predictions of the domains $d^{(i)}$. This can be formulated as a joint optimization problem,

$$\min_{\boldsymbol{\theta}_g \boldsymbol{\theta}_y, \boldsymbol{\theta}_d} \sum_{i=1}^{N_\ell + N_u} \ell_d(\boldsymbol{g}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_g), d^{(i)}; \boldsymbol{\theta}_d) - \sum_{i=1}^{N_\ell} \ell_y(\boldsymbol{g}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_g), y^{(i)}; \boldsymbol{\theta}_y), \tag{5.42}$$

where N_ℓ is the number of labeled instances and N_u is the number of unlabeled instances, with the labeled instances appearing first in the dataset. This setup is shown in Figure 5.4. The loss can be optimized by stochastic gradient descent, jointly training the parameters of the non-linear transformation $\boldsymbol{\theta}_g$, and the parameters of the prediction models $\boldsymbol{\theta}_d$ and $\boldsymbol{\theta}_y$.

5.5 *Other approaches to learning with latent variables

Expectation-maximization provides a general approach to learning with latent variables, but it has limitations. One is the sensitivity to initialization; in practical applications, considerable attention may need to be devoted to finding a good initialization. A second issue is that EM tends to be easiest to apply in cases where the latent variables have a clear decomposition (in the cases we have considered, they decompose across the instances). For these reasons, it is worth briefly considering some alternatives to EM.

5.5.1 Sampling

In EM clustering, there is a distribution $\boldsymbol{q}^{(i)}$ for the missing data related to each instance. The M-step consists of updating the parameters of this distribution. An alternative is to draw samples of the latent variables. If the sampling distribution is designed correctly, this procedure will eventually converge to drawing samples from the true posterior over the missing data, $\text{p}(\boldsymbol{z}^{(1:N_z)} \mid \boldsymbol{x}^{(1:N_x)})$. For example, in the case of clustering, the missing data $\boldsymbol{z}^{(1:N_z)}$ is the set of cluster memberships, $\boldsymbol{y}^{(1:N)}$, so we draw samples from the posterior distribution over clusterings of the data. If a single clustering is required, we can select the one with the highest conditional likelihood, $\hat{\boldsymbol{z}} = \text{argmax}_{\boldsymbol{z}} \text{p}(\boldsymbol{z}^{(1:N_z)} \mid \boldsymbol{x}^{(1:N_x)})$.

This general family of algorithms is called **Markov Chain Monte Carlo (MCMC)**: "Monte Carlo" because it is based on a series of random draws; "Markov Chain" because the sampling procedure must be designed such that each sample depends only on the previous sample, and not on the entire sampling history. **Gibbs sampling** is an MCMC algorithm in which each latent variable is sampled from its posterior distribution,

$$z^{(n)} \mid \boldsymbol{x}, \boldsymbol{z}^{(-n)} \sim \text{p}(z^{(n)} \mid \boldsymbol{x}, \boldsymbol{z}^{(-n)}), \tag{5.43}$$

where $\boldsymbol{z}^{(-n)}$ indicates $\{\boldsymbol{z} \backslash z^{(n)}\}$, the set of all latent variables except for $z^{(n)}$. Repeatedly drawing samples over all latent variables constructs a Markov chain that is guaranteed to converge to a sequence of samples from $\text{p}(\boldsymbol{z}^{(1:N_z)} \mid \boldsymbol{x}^{(1:N_x)})$. In probabilistic clustering, the sampling distribution has the following form,

$$\text{p}(z^{(i)} \mid \boldsymbol{x}, \boldsymbol{z}^{(-i)}) = \frac{\text{p}(\boldsymbol{x}^{(i)} \mid z^{(i)}; \boldsymbol{\phi}) \times \text{p}(z^{(i)}; \boldsymbol{\mu})}{\sum_{z=1}^{K} \text{p}(\boldsymbol{x}^{(i)} \mid z; \boldsymbol{\phi}) \times \text{p}(z; \boldsymbol{\mu})} \tag{5.44}$$

$$\propto \text{Multinomial}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{z^{(i)}}) \times \boldsymbol{\mu}_{z^{(i)}}. \tag{5.45}$$

In this case, the sampling distribution does not depend on the other instances: the posterior distribution over each $z^{(i)}$ can be computed from $\boldsymbol{x}^{(i)}$ and the parameters given the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$.

In sampling algorithms, there are several choices for how to deal with the parameters. One possibility is to sample them too. To do this, we must add them to the generative story, by introducing a prior distribution. For the multinomial and categorical parameters in the EM clustering model, the **Dirichlet distribution** is a typical choice, since it defines a probability on exactly the set of vectors that can be parameters: vectors that sum to one and include only non-negative numbers.[10]

To incorporate this prior, the generative model must be augmented to indicate that each $\boldsymbol{\phi}_z \sim \text{Dirichlet}(\boldsymbol{\alpha}_\phi)$, and $\boldsymbol{\mu} \sim \text{Dirichlet}(\boldsymbol{\alpha}_\mu)$. The hyperparameters $\boldsymbol{\alpha}$ are typically set to a constant vector $\boldsymbol{\alpha} = [\alpha, \alpha, \ldots, \alpha]$. When α is large, the Dirichlet distribution tends to

[10] If $\sum_i^K \theta_i = 1$ and $\theta_i \geq 0$ for all i, then $\boldsymbol{\theta}$ is said to be on the $K-1$ **simplex**. A Dirichlet distribution with

generate vectors that are nearly uniform; when α is small, it tends to generate vectors that assign most of their probability mass to a few entries. Given prior distributions over $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$, we can now include them in Gibbs sampling, drawing values for these parameters from posterior distributions that are conditioned on the other variables in the model.

Unfortunately, sampling $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$ usually leads to slow "mixing", meaning that adjacent samples tend to be similar, so that a large number of samples is required to explore the space of random variables. The reason is that the sampling distributions for the parameters are tightly constrained by the cluster memberships $\boldsymbol{y}^{(i)}$, which in turn are tightly constrained by the parameters. There are two solutions that are frequently employed:

- **Empirical Bayesian** methods maintain $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$ as parameters rather than latent variables. They still employ sampling in the E-step of the EM algorithm, but they update the parameters using expected counts that are computed from the samples rather than from parametric distributions. This EM-MCMC hybrid is also known as Monte Carlo Expectation Maximization (MCEM; Wei and Tanner, 1990), and is well-suited for cases in which it is difficult to compute $\boldsymbol{q}^{(i)}$ directly.
- In **collapsed Gibbs sampling**, we analytically integrate $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$ out of the model. The cluster memberships $y^{(i)}$ are the only remaining latent variable; we sample them from the compound distribution,

$$\text{p}(y^{(i)} \mid \boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(-i)}; \alpha_\phi, \alpha_\mu) = \int_{\boldsymbol{\phi},\boldsymbol{\mu}} \text{p}(\boldsymbol{\phi}, \boldsymbol{\mu} \mid \boldsymbol{y}^{(-i)}, \boldsymbol{x}^{(1:N)}; \alpha_\phi, \alpha_\mu) \text{p}(y^{(i)} \mid \boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(-i)}, \boldsymbol{\phi}, \boldsymbol{\mu}) d\boldsymbol{\phi} d\boldsymbol{\mu}. \quad [5.48]$$

  For multinomial and Dirichlet distributions, this integral can be computed in closed form.

MCMC algorithms are guaranteed to converge to the true posterior distribution over the latent variables, but there is no way to know how long this will take. In practice, the rate of convergence depends on initialization, just as expectation-maximization depends on initialization to avoid local optima. Thus, while Gibbs Sampling and other MCMC algorithms provide a powerful and flexible array of techniques for statistical inference in latent variable models, they are not a panacea for the problems experienced by EM.

---

parameter $\boldsymbol{\alpha} \in \mathbb{R}_+^K$ has support over the $K-1$ simplex,

$$\text{p}_{\text{Dirichlet}}(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{K} \theta_i^{\alpha_i - 1} \quad [5.46]$$

$$B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K} \alpha_i)}, \quad [5.47]$$

with $\Gamma(\cdot)$ indicating the gamma function, a generalization of the factorial function to non-negative reals.

5.5.2 Spectral learning

Another approach to learning with latent variables is based on the **method of moments**, which makes it possible to avoid the problem of non-convex log-likelihood. Write $\overline{\boldsymbol{x}}^{(i)}$ for the normalized vector of word counts in document i, so that $\overline{\boldsymbol{x}}^{(i)} = \boldsymbol{x}^{(i)} / \sum_{j=1}^{V} x_j^{(i)}$. Then we can form a matrix of word-word co-occurrence probabilities,

$$\mathbf{C} = \sum_{i=1}^{N} \overline{\boldsymbol{x}}^{(i)} (\overline{\boldsymbol{x}}^{(i)})^\top. \tag{5.49}$$

The expected value of this matrix under $\text{p}(\boldsymbol{x} \mid \boldsymbol{\phi}, \mu)$, as

$$E[\mathbf{C}] = \sum_{i=1}^{N} \sum_{k=1}^{K} \Pr(Z^{(i)} = k; \boldsymbol{\mu}) \boldsymbol{\phi}_k \boldsymbol{\phi}_k^\top \tag{5.50}$$

$$= \sum_{k}^{K} N \mu_k \boldsymbol{\phi}_k \boldsymbol{\phi}_k^\top \tag{5.51}$$

$$= \Phi \text{Diag}(N\mu) \Phi^\top, \tag{5.52}$$

where Φ is formed by horizontally concatenating $\boldsymbol{\phi}_1 \ldots \boldsymbol{\phi}_K$, and $\text{Diag}(N\mu)$ indicates a diagonal matrix with values $N\mu_k$ at position (k, k). Setting $\mathbf{C}$ equal to its expectation gives,

$$\mathbf{C} = \boldsymbol{\Phi} \text{Diag}(N\mu) \boldsymbol{\Phi}^\top, \tag{5.53}$$

which is similar to the eigendecomposition $\mathbf{C} = \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^\top$. This suggests that simply by finding the eigenvectors and eigenvalues of $\mathbf{C}$, we could obtain the parameters $\boldsymbol{\phi}$ and μ, and this is what motivates the name **spectral learning**.

While moment-matching and eigendecomposition are similar in form, they impose different constraints on the solutions: eigendecomposition requires orthonormality, so that $\mathbf{Q}\mathbf{Q}^\top = \mathbb{I}$; in estimating the parameters of a text clustering model, we require that $\boldsymbol{\mu}$ and the columns of Φ are probability vectors. Spectral learning algorithms must therefore include a procedure for converting the solution into vectors that are non-negative and sum to one. One approach is to replace eigendecomposition (or the related singular value decomposition) with non-negative matrix factorization (Xu et al., 2003), which guarantees that the solutions are non-negative (Arora et al., 2013).

After obtaining the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$, the distribution over clusters can be computed from Bayes' rule:

$$\text{p}(z^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\phi}, \boldsymbol{\mu}) \propto \text{p}(\boldsymbol{x}^{(i)} \mid z^{(i)}; \boldsymbol{\phi}) \times \text{p}(z^{(i)}; \mu). \tag{5.54}$$

Spectral learning yields provably good solutions without regard to initialization, and can be quite fast in practice. However, it is more difficult to apply to a broad family of generative models than EM and Gibbs Sampling. For more on applying spectral learning across a range of latent variable models, see Anandkumar et al. (2014).

Additional resources

There are a number of other learning paradigms that deviate from supervised learning.

- **Active learning**: the learner selects unlabeled instances and requests annotations (Settles, 2012).
- **Multiple instance learning**: labels are applied to bags of instances, with a positive label applied if at least one instance in the bag meets the criterion (Dietterich et al., 1997; Maron and Lozano-Pérez, 1998).
- **Constraint-driven learning**: supervision is provided in the form of explicit constraints on the learner (Chang et al., 2007; Ganchev et al., 2010).
- **Distant supervision**: noisy labels are generated from an external resource (Mintz et al., 2009, also see § 17.2.3).
- **Multitask learning**: the learner induces a representation that can be used to solve multiple classification tasks (Collobert et al., 2011).
- **Transfer learning**: the learner must solve a classification task that differs from the labeled data (Pan and Yang, 2010).

Expectation-maximization was introduced by Dempster et al. (1977), and is discussed in more detail by Murphy (2012). Like most machine learning treatments, Murphy focuses on continuous observations and Gaussian likelihoods, rather than the discrete observations typically encountered in natural language processing. Murphy (2012) also includes an excellent chapter on MCMC; for a textbook-length treatment, see Robert and Casella (2013). For still more on Bayesian latent variable models, see Barber (2012), and for applications of Bayesian models to natural language processing, see Cohen (2016). Surveys are available for semi-supervised learning (Zhu and Goldberg, 2009) and domain adaptation (Søgaard, 2013), although both pre-date the current wave of interest in deep learning.

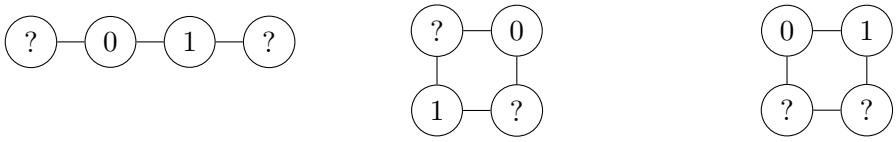
Exercises

1. Derive the expectation maximization update for the parameter $\boldsymbol{\mu}$ in the EM clustering model.

2. Derive the E-step and M-step updates for the following generative model. You may assume that the labels $y^{(i)}$ are observed, but $z_m^{(i)}$ is not.

   - For each instance i,
     - Draw label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$
     - For each token $m \in \{1, 2, \ldots, M^{(i)}\}$
       * Draw $z_m^{(i)} \sim \text{Categorical}(\pi)$
       * If $z_m^{(i)} = 0$, draw the current token from a label-specific distribution, $w_m^{(i)} \sim \boldsymbol{\phi}_{y^{(i)}}$
       * If $z_m^{(i)} = 1$, draw the current token from a document-specific distribution, $w_m^{(i)} \sim \boldsymbol{\nu}^{(i)}$

3. Using the iterative updates in Equations 5.34-5.36, compute the outcome of the label propagation algorithm for the following examples.



   The value inside the node indicates the label, $y^{(i)} \in \{0, 1\}$, with $y^{(i)} =?$ for unlabeled nodes. The presence of an edge between two nodes indicates $w_{i,j} = 1$, and the absence of an edge indicates $w_{i,j} = 0$. For the third example, you need only compute the first three iterations, and then you can guess at the solution in the limit.

4. Use expectation-maximization clustering to train a word-sense induction system, applied to the word *say*.

   - Import NLTK, run NLTK.DOWNLOAD() and select SEMCOR. Import SEMCOR from NLTK.CORPUS.
   - The command SEMCOR.TAGGED_SENTENCES(TAG='SENSE') returns an iterator over sense-tagged sentences in the corpus. Each sentence can be viewed as an iterator over TREE objects. For TREE objects that are sense-annotated words, you can access the annotation as TREE.LABEL(), and the word itself with TREE.LEAVES(). So SEMCOR.TAGGED_SENTENCES(TAG='SENSE')[0][2].LABEL() would return the sense annotation of the third word in the first sentence.
   - Extract all sentences containing the senses SAY.V.01 and SAY.V.02.
   - Build bag-of-words vectors $\boldsymbol{x}^{(i)}$, containing the counts of other words in those sentences, including all words that occur in at least two sentences.
   - Implement and run expectation-maximization clustering on the merged data.

- Compute the frequency with which each cluster includes instances of SAY.V.01 and SAY.V.02.

In the remaining exercises, you will try out some approaches for semisupervised learning and domain adaptation. You will need datasets in multiple domains. You can obtain product reviews in multiple domains here: `https://www.cs.jhu.edu/~mdredze/datasets/sentiment/processed_acl.tar.gz`. Choose a source and target domain, e.g. dvds and books, and divide the data for the target domain into training and test sets of equal size.

5. First, quantify the cost of cross-domain transfer.

   - Train a logistic regression classifier on the source domain training set, and evaluate it on the target domain test set.
   - Train a logistic regression classifier on the target domain training set, and evaluate it on the target domain test set. This it the "direct transfer" baseline.

   Compute the difference in accuracy, which is a measure of the transfer loss across domains.

6. Next, apply the **label propagation** algorithm from § 5.3.2.

   As a baseline, using only 5% of the target domain training set, train a classifier, and compute its accuracy on the target domain test set.

   Next, apply label propagation:

   - Compute the label matrix $\mathbf{Q}_L$ for the labeled data (5% of the target domain training set), with each row equal to an indicator vector for the label (positive or negative).
   - Iterate through the target domain instances, including both test and training data. At each instance i, compute all w_{ij}, using Equation 5.32, with $\alpha = 0.01$. Use these values to fill in column i of the transition matrix $\mathbf{T}$, setting all but the ten largest values to zero for each column i. Be sure to normalize the column so that the remaining values sum to one. You may need to use a sparse matrix for this to fit into memory.
   - Apply the iterative updates from Equations 5.34-5.36 to compute the outcome of the label propagation algorithm for the unlabeled examples.

   Select the test set instances from $\mathbf{Q}_U$, and compute the accuracy of this method. Compare with the supervised classifier trained only on the 5% sample of the target domain training set.

7. Using only 5% of the target domain training data (and all of the source domain training data), implement one of the supervised domain adaptation baselines in § 5.4.1. See if this improves on the "direct transfer" baseline from the previous problem

8. Implement EASYADAPT (§ 5.4.1), again using 5% of the target domain training data and all of the source domain data.

9. Now try unsupervised domain adaptation, using the "linear projection" method described in § 5.4.2. Specifically:

   - Identify 500 pivot features as the words with the highest frequency in the (complete) training data for the source and target domains. Specifically, let x_i^d be the count of the word i in domain d: choose the 500 words with the largest values of $\min(x_i^{\text{source}}, x_i^{\text{target}})$.
   - Train a classifier to predict each pivot feature from the remaining words in the document.
   - Arrange the features of these classifiers into a matrix $\mathbf{\Phi}$, and perform truncated singular value decomposition, with $k = 20$
   - Train a classifier from the source domain data, using the combined features $\boldsymbol{x}^{(i)} \oplus \mathbf{U}^\top \boldsymbol{x}^{(i)}$ — these include the original bag-of-words features, plus the projected features.
   - Apply this classifier to the target domain test set, and compute the accuracy.

Part II

Sequences and trees

Chapter 6

Language models

In probabilistic classification, the problem is to compute the probability of a label, conditioned on the text. Let's now consider the inverse problem: computing the probability of text itself. Specifically, we will consider models that assign probability to a sequence of word tokens, $\text{p}(w_1, w_2, \ldots, w_M)$, with $w_m \in \mathcal{V}$. The set $\mathcal{V}$ is a discrete vocabulary,

$$\mathcal{V} = \{\textit{aardvark}, \textit{abacus}, \ldots, \textit{zither}\}. \quad [6.1]$$

Why would you want to compute the probability of a word sequence? In many applications, the goal is to produce word sequences as output:

- In **machine translation** (chapter 18), we convert from text in a source language to text in a target language.
- In **speech recognition**, we convert from audio signal to text.
- In **summarization** (§ 16.3.4; § 19.2), we convert from long texts into short texts.
- In **dialogue systems** (§ 19.3), we convert from the user's input (and perhaps an external knowledge base) into a text response.

In many of the systems for performing these tasks, there is a subcomponent that computes the probability of the output text. The purpose of this component is to generate texts that are more **fluent**. For example, suppose we want to translate a sentence from Spanish to English.

(6.1) El cafe negro me gusta mucho.

Here is a literal word-for-word translation (a **gloss**):

(6.2) The coffee black me pleases much.

A good language model of English will tell us that the probability of this translation is low, in comparison with more grammatical alternatives,

$$\text{p}(\textit{The coffee black me pleases much}) < \text{p}(\textit{I love dark coffee}). \quad [6.2]$$

How can we use this fact? Warren Weaver, one of the early leaders in machine translation, viewed it as a problem of breaking a secret code (Weaver, 1955):

> When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.'

This observation motivates a generative model (like Naïve Bayes):

- The English sentence $\boldsymbol{w}^{(e)}$ is generated from a **language model**, $\text{p}_e(\boldsymbol{w}^{(e)})$.
- The Spanish sentence $\boldsymbol{w}^{(s)}$ is then generated from a **translation model**, $\text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)})$.

Given these two distributions, translation can be performed by Bayes' rule:

$$\text{p}_{e|s}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}) \propto \text{p}_{e,s}(\boldsymbol{w}^{(e)}, \boldsymbol{w}^{(s)}) \quad [6.3]$$

$$= \text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) \times \text{p}_e(\boldsymbol{w}^{(e)}). \quad [6.4]$$

This is sometimes called the **noisy channel model**, because it envisions English text turning into Spanish by passing through a noisy channel, $\text{p}_{s|e}$. What is the advantage of modeling translation this way, as opposed to modeling $\text{p}_{e|s}$ directly? The crucial point is that the two distributions $\text{p}_{s|e}$ (the translation model) and p_e (the language model) can be estimated from separate data. The translation model requires examples of correct translations, but the language model requires only text in English. Such monolingual data is much more widely available. Furthermore, once estimated, the language model p_e can be reused in any application that involves generating English text, including translation from other languages.

6.1 N-gram language models

A simple approach to computing the probability of a sequence of tokens is to use a **relative frequency estimate**. Consider the quote, attributed to Picasso, "*computers are useless, they can only give you answers.*" One way to estimate the probability of this sentence is,

$$\begin{aligned}&\text{p}(\textit{Computers are useless, they can only give you answers}) \\ &= \frac{\text{count}(\textit{Computers are useless, they can only give you answers})}{\text{count}(\text{all sentences ever spoken})}\end{aligned} \quad [6.5]$$

This estimator is **unbiased**: in the theoretical limit of infinite data, the estimate will be correct. But in practice, we are asking for accurate counts over an infinite number of events, since sequences of words can be arbitrarily long. Even with an aggressive upper bound of, say, $M = 20$ tokens in the sequence, the number of possible sequences is V^{20}, where $V = |\mathcal{V}|$. A small vocabularly for English would have $V = 10^5$, so there are 10^{100} possible sequences. Clearly, this estimator is very data-hungry, and suffers from high variance: even grammatical sentences will have probability zero if they have not occurred in the training data.[1] We therefore need to introduce bias to have a chance of making reliable estimates from finite training data. The language models that follow in this chapter introduce bias in various ways.

We begin with n-gram language models, which compute the probability of a sequence as the product of probabilities of subsequences. The probability of a sequence $\text{p}(\boldsymbol{w}) = \text{p}(w_1, w_2, \ldots, w_M)$ can be refactored using the chain rule (see § A.2):

$$\begin{aligned}\text{p}(\boldsymbol{w}) =& \text{p}(w_1, w_2, \ldots, w_M) && [6.6]\\ =& \text{p}(w_1) \times \text{p}(w_2 \mid w_1) \times \text{p}(w_3 \mid w_2, w_1) \times \ldots \times \text{p}(w_M \mid w_{M-1}, \ldots, w_1) && [6.7]\end{aligned}$$

Each element in the product is the probability of a word given all its predecessors. We can think of this as a *word prediction* task: given the context *Computers are*, we want to compute a probability over the next token. The relative frequency estimate of the probability of the word *useless* in this context is,

$$\begin{aligned}\text{p}(\textit{useless} \mid \textit{computers are}) &= \frac{\text{count}(\textit{computers are useless})}{\sum_{x \in \mathcal{V}} \text{count}(\textit{computers are } x)} \\ &= \frac{\text{count}(\textit{computers are useless})}{\text{count}(\textit{computers are})}.\end{aligned}$$

We haven't made any approximations yet, and we could have just as well applied the chain rule in reverse order,

$$\text{p}(\boldsymbol{w}) = \text{p}(w_M) \times \text{p}(w_{M-1} \mid w_M) \times \ldots \times \text{p}(w_1 \mid w_2, \ldots, w_M), \quad [6.8]$$

or in any other order. But this means that we also haven't really made any progress: to compute the conditional probability $\text{p}(w_M \mid w_{M-1}, w_{M-2}, \ldots, w_1)$, we would need to model V^{M-1} contexts. Such a distribution cannot be estimated from any realistic sample of text.

To solve this problem, n-gram models make a crucial simplifying approximation: they condition on only the past $n - 1$ words.

$$\text{p}(w_m \mid w_{m-1} \ldots w_1) \approx \text{p}(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \quad [6.9]$$

[1] Chomsky famously argued that this is evidence against the very concept of probabilistic language models: no such model could distinguish the grammatical sentence *colorless green ideas sleep furiously* from the ungrammatical permutation *furiously sleep ideas green colorless*.

This means that the probability of a sentence $\boldsymbol{w}$ can be approximated as

$$\mathrm{p}(w_1, \ldots, w_M) \approx \prod_{m=1}^{M} \mathrm{p}(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \quad [6.10]$$

To compute the probability of an entire sentence, it is convenient to pad the beginning and end with special symbols $\square$ and $\blacksquare$. Then the bigram ($n = 2$) approximation to the probability of *I like black coffee* is:

$$\mathrm{p}(\textit{I like black coffee}) = \mathrm{p}(I \mid \square) \times \mathrm{p}(\textit{like} \mid I) \times \mathrm{p}(\textit{black} \mid \textit{like}) \times \mathrm{p}(\textit{coffee} \mid \textit{black}) \times \mathrm{p}(\blacksquare \mid \textit{coffee}). \quad [6.11]$$

This model requires estimating and storing the probability of only V^n events, which is exponential in the order of the n-gram, and not V^M, which is exponential in the length of the sentence. The n-gram probabilities can be computed by relative frequency estimation,

$$\mathrm{p}(w_m \mid w_{m-1}, w_{m-2}) = \frac{\text{count}(w_{m-2}, w_{m-1}, w_m)}{\sum_{w'} \text{count}(w_{m-2}, w_{m-1}, w')} \quad [6.12]$$

The hyperparameter n controls the size of the context used in each conditional probability. If this is misspecified, the language model will perform poorly. Let's consider the potential problems concretely.

When n is too small. Consider the following sentences:

(6.3) **Gorillas** always like to groom **their** friends.

(6.4) The **computer** that's on the 3rd floor of our office building **crashed**.

In each example, the words written in bold depend on each other: the likelihood of *their* depends on knowing that *gorillas* is plural, and the likelihood of *crashed* depends on knowing that the subject is a *computer*. If the n-grams are not big enough to capture this context, then the resulting language model would offer probabilities that are too low for these sentences, and too high for sentences that fail basic linguistic tests like number agreement.

When n is too big. In this case, it is hard good estimates of the n-gram parameters from our dataset, because of data sparsity. To handle the *gorilla* example, it is necessary to model 6-grams, which means accounting for V^6 events. Under a very small vocabulary of $V = 10^4$, this means estimating the probability of 10^{24} distinct events.

These two problems point to another **bias-variance tradeoff** (see § 2.2.4). A small n-gram size introduces high bias, and a large n-gram size introduces high variance. We can even have both problems at the same time! Language is full of long-range dependencies that we cannot capture because n is too small; at the same time, language datasets are full of rare phenomena, whose probabilities we fail to estimate accurately because n is too large. One solution is to try to keep n large, while still making low-variance estimates of the underlying parameters. To do this, we will introduce a different sort of bias: **smoothing**.

6.2 Smoothing and discounting

Limited data is a persistent problem in estimating language models. In § 6.1, we presented n-grams as a partial solution. Bit sparse data can be a problem even for low-order n-grams; at the same time, many linguistic phenomena, like subject-verb agreement, cannot be incorporated into language models without high-order n-grams. It is therefore necessary to add additional inductive biases to n-gram language models. This section covers some of the most intuitive and common approaches, but there are many more (see Chen and Goodman, 1999).

6.2.1 Smoothing

A major concern in language modeling is to avoid the situation $\text{p}(\boldsymbol{w}) = 0$, which could arise as a result of a single unseen n-gram. A similar problem arose in Naïve Bayes, and the solution was **smoothing**: adding imaginary "pseudo" counts. The same idea can be applied to n-gram language models, as shown here in the bigram case,

$$\text{p}_{\text{smooth}}(w_m \mid w_{m-1}) = \frac{\text{count}(w_{m-1}, w_m) + \alpha}{\sum_{w' \in \mathcal{V}} \text{count}(w_{m-1}, w') + V\alpha}. \quad [6.13]$$

This basic framework is called **Lidstone smoothing**, but special cases have other names:

- **Laplace smoothing** corresponds to the case $\alpha = 1$.
- **Jeffreys-Perks law** corresponds to the case $\alpha = 0.5$, which works well in practice and benefits from some theoretical justification (Manning and Schütze, 1999).

To ensure that the probabilities are properly normalized, anything that we add to the numerator (α) must also appear in the denominator ($V\alpha$). This idea is reflected in the concept of **effective counts**:

$$c_i^* = (c_i + \alpha)\frac{M}{M + V\alpha}, \quad [6.14]$$

			Lidstone smoothing, $\alpha = 0.1$		Discounting, $d = 0.1$	
	counts	unsmoothed probability	effective counts	smoothed probability	effective counts	smoothed probability
impropriety	8	0.4	7.826	0.391	7.9	0.395
offense	5	0.25	4.928	0.246	4.9	0.245
damage	4	0.2	3.961	0.198	3.9	0.195
deficiencies	2	0.1	2.029	0.101	1.9	0.095
outbreak	1	0.05	1.063	0.053	0.9	0.045
infirmity	0	0	0.097	0.005	0.25	0.013
cephalopods	0	0	0.097	0.005	0.25	0.013

Table 6.1: Example of Lidstone smoothing and absolute discounting in a bigram language model, for the context $(\textit{alleged}, \_)$, for a toy corpus with a total of twenty counts over the seven words shown. Note that discounting decreases the probability for all but the unseen words, while Lidstone smoothing increases the effective counts and probabilities for *deficiencies* and *outbreak*.

where c_i is the count of event i, c_i^* is the effective count, and $M = \sum_{i=1}^{V} c_i$ is the total number of tokens in the dataset $(w_1, w_2, \ldots, w_M)$. This term ensures that $\sum_{i=1}^{V} c_i^* = \sum_{i=1}^{V} c_i = M$. The **discount** for each n-gram is then computed as,

$$d_i = \frac{c_i^*}{c_i} = \frac{(c_i + \alpha)}{c_i} \frac{M}{(M + V\alpha)}.$$

6.2.2 Discounting and backoff

Discounting "borrows" probability mass from observed n-grams and redistributes it. In Lidstone smoothing, the borrowing is done by increasing the denominator of the relative frequency estimates. The borrowed probability mass is then redistributed by increasing the numerator for all n-grams. Another approach would be to borrow the same amount of probability mass from all observed n-grams, and redistribute it among only the unobserved n-grams. This is called **absolute discounting**. For example, suppose we set an absolute discount $d = 0.1$ in a bigram model, and then redistribute this probability mass equally over the unseen words. The resulting probabilities are shown in Table 6.1.

Discounting reserves some probability mass from the observed data, and we need not redistribute this probability mass equally. Instead, we can **backoff** to a lower-order language model: if you have trigrams, use trigrams; if you don't have trigrams, use bigrams; if you don't even have bigrams, use unigrams. This is called **Katz backoff**. In the simple

case of backing off from bigrams to unigrams, the bigram probabilities are,

$$c^*(i,j) = c(i,j) - d \quad [6.15]$$

$$\mathrm{p}_{\text{Katz}}(i \mid j) = \begin{cases} \frac{c^*(i,j)}{c(j)} & \text{if } c(i,j) > 0 \\ \alpha(j) \times \frac{\mathrm{p}_{\text{unigram}}(i)}{\sum_{i':c(i',j)=0} \mathrm{p}_{\text{unigram}}(i')} & \text{if } c(i,j) = 0. \end{cases} \quad [6.16]$$

The term $\alpha(j)$ indicates the amount of probability mass that has been discounted for context j. This probability mass is then divided across all the unseen events, $\{i' : c(i', j) = 0\}$, proportional to the unigram probability of each word i'. The discount parameter d can be optimized to maximize performance (typically held-out log-likelihood) on a development set.

6.2.3 *Interpolation

Backoff is one way to combine different order n-gram models. An alternative approach is **interpolation**: setting the probability of a word in context to a weighted sum of its probabilities across progressively shorter contexts.

Instead of choosing a single n for the size of the n-gram, we can take the weighted average across several n-gram probabilities. For example, for an interpolated trigram model,

$$\begin{aligned} \mathrm{p}_{\text{Interpolation}}(w_m \mid w_{m-1}, w_{m-2}) = &\lambda_3 \mathrm{p}_3^*(w_m \mid w_{m-1}, w_{m-2}) \\ &+ \lambda_2 \mathrm{p}_2^*(w_m \mid w_{m-1}) \\ &+ \lambda_1 \mathrm{p}_1^*(w_m). \end{aligned}$$

In this equation, p_n^* is the unsmoothed empirical probability given by an n-gram language model, and λ_n is the weight assigned to this model. To ensure that the interpolated $\mathrm{p}(\boldsymbol{w})$ is still a valid probability distribution, the values of λ must obey the constraint, $\sum_{n=1}^{n_{\max}} \lambda_n = 1$. But how to find the specific values?

An elegant solution is **expectation-maximization**. Recall from chapter 5 that we can think about EM as learning with *missing data*: we just need to choose missing data such that learning would be easy if it weren't missing. What's missing in this case? Think of each word w_m as drawn from an n-gram of unknown size, $z_m \in \{1 \ldots n_{\max}\}$. This z_m is the missing data that we are looking for. Therefore, the application of EM to this problem involves the following **generative model**:

for Each token $w_m, m = 1, 2, \ldots, M$ **do**:
  draw the n-gram size $z_m \sim$ Categorical(λ);
  draw $w_m \sim \mathrm{p}_{z_m}^*(w_m \mid w_{m-1}, \ldots, w_{m-z_m})$.

If the missing data $\{Z_m\}$ were known, then λ could be estimated as the relative frequency,

$$\lambda_z = \frac{\text{count}(Z_m = z)}{M} \tag{6.17}$$

$$\propto \sum_{m=1}^{M} \delta(Z_m = z). \tag{6.18}$$

But since we do not know the values of the latent variables Z_m, we impute a distribution q_m in the E-step, which represents the degree of belief that word token w_m was generated from a n-gram of order z_m,

$$q_m(z) \triangleq \Pr(Z_m = z \mid \boldsymbol{w}_{1:m}; \lambda) \tag{6.19}$$

$$= \frac{\text{p}(w_m \mid \boldsymbol{w}_{1:m-1}, Z_m = z) \times \text{p}(z)}{\sum_{z'} \text{p}(w_m \mid \boldsymbol{w}_{1:m-1}, Z_m = z') \times \text{p}(z')} \tag{6.20}$$

$$\propto \text{p}^*_z(w_m \mid \boldsymbol{w}_{1:m-1}) \times \lambda_z. \tag{6.21}$$

In the M-step, λ is computed by summing the expected counts under q,

$$\lambda_z \propto \sum_{m=1}^{M} q_m(z). \tag{6.22}$$

A solution is obtained by iterating between updates to q and λ. The complete algorithm is shown in Algorithm 10.

Algorithm 10 Expectation-maximization for interpolated language modeling

procedure ESTIMATE INTERPOLATED n-GRAM ($\boldsymbol{w}_{1:M}$, $\{\text{p}^*_n\}_{n \in 1:n_{\text{max}}}$)
  for $z \in \{1, 2, \ldots, n_{\text{max}}\}$ **do** ▷ Initialization
    $\lambda_z \leftarrow \frac{1}{n_{\text{max}}}$
  repeat
    for $m \in \{1, 2, \ldots, M\}$ **do** ▷ E-step
      for $z \in \{1, 2, \ldots, n_{\text{max}}\}$ **do**
        $q_m(z) \leftarrow \text{p}^*_z(w_m \mid \boldsymbol{w}_{1:m-}) \times \lambda_z$
      $\boldsymbol{q}_m \leftarrow \text{Normalize}(\boldsymbol{q}_m)$
    for $z \in \{1, 2, \ldots, n_{\text{max}}\}$ **do** ▷ M-step
      $\lambda_z \leftarrow \frac{1}{M} \sum_{m=1}^{M} q_m(z)$
  until tired
  return $\boldsymbol{\lambda}$

6.2.4 *Kneser-Ney smoothing

Kneser-Ney smoothing is based on absolute discounting, but it redistributes the resulting probability mass in a different way from Katz backoff. Empirical evidence points to Kneser-Ney smoothing as the state-of-art for n-gram language modeling (Goodman, 2001). To motivate Kneser-Ney smoothing, consider the example: *I recently visited* _. Which of the following is more likely: *Francisco* or *Duluth*?

Now suppose that both bigrams *visited Duluth* and *visited Francisco* are unobserved in the training data, and furthermore, that the unigram probability $\text{p}_1^*(\textit{Francisco})$ is greater than $\text{p}_1^*(\textit{Duluth})$. Nonetheless we would still guess that $\text{p}(\textit{visited Duluth}) > \text{p}(\textit{visited Francisco})$, because *Duluth* is a more "versatile" word: it can occur in many contexts, while *Francisco* usually occurs in a single context, following the word *San*. This notion of versatility is the key to Kneser-Ney smoothing.

Writing u for a context of undefined length, and $\text{count}(w, u)$ as the count of word w in context u, we define the Kneser-Ney bigram probability as

$$\text{p}_{KN}(w \mid u) = \begin{cases} \frac{\max(\text{count}(w,u)-d,0)}{\text{count}(u)}, & \text{count}(w,u) > 0 \\ \alpha(u) \times \text{p}_{\text{continuation}}(w), & \text{otherwise} \end{cases} \quad [6.23]$$

$$\text{p}_{\text{continuation}}(w) = \frac{|u : \text{count}(w,u) > 0|}{\sum_{w' \in \mathcal{V}} |u' : \text{count}(w',u') > 0|}. \quad [6.24]$$

Probability mass using absolute discounting d, which is taken from all unobserved n-grams. The total amount of discounting in context u is $d \times |w : \text{count}(w,u) > 0|$, and we divide this probability mass among the unseen n-grams. To account for versatility, we define the *continuation probability* $\text{p}_{\text{continuation}}(w)$ as proportional to the number of observed contexts in which w appears. The numerator of the continuation probability is the number of contexts u in which w appears; the denominator normalizes the probability by summing the same quantity over all words w'. The coefficient $\alpha(u)$ is set to ensure that the probability distribution $\text{p}_{KN}(w \mid u)$ sums to one over the vocabulary w.

The idea of modeling versatility by counting contexts may seem heuristic, but there is an elegant theoretical justification from Bayesian nonparametrics (Teh, 2006). Kneser-Ney smoothing on n-grams was the dominant language modeling technique before the arrival of neural language models.

6.3 Recurrent neural network language models

N-gram language models have been largely supplanted by neural networks. These models do not make the n-gram assumption of restricted context; indeed, they can incorporate arbitrarily distant contextual information, while remaining computationally and statistically tractable.

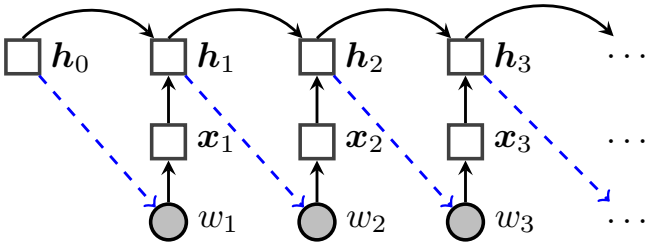


Figure 6.1: The recurrent neural network language model, viewed as an "unrolled" computation graph. Solid lines indicate direct computation, dotted blue lines indicate probabilistic dependencies, circles indicate random variables, and squares indicate computation nodes.

The first insight behind neural language models is to treat word prediction as a *discriminative* learning task.[2] The goal is to compute the probability $\text{p}(w \mid u)$, where $w \in \mathcal{V}$ is a word, and u is the context, which depends on the previous words. Rather than directly estimating the word probabilities from (smoothed) relative frequencies, we can treat treat language modeling as a machine learning problem, and estimate parameters that maximize the log conditional probability of a corpus.

The second insight is to reparametrize the probability distribution $\text{p}(w \mid u)$ as a function of two dense K-dimensional numerical vectors, $\boldsymbol{\beta}_w \in \mathbb{R}^K$, and $\boldsymbol{v}_u \in \mathbb{R}^K$,

$$\text{p}(w \mid u) = \frac{\exp(\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{v}_u)}, \quad [6.25]$$

where $\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u$ represents a dot product. As usual, the denominator ensures that the probability distribution is properly normalized. This vector of probabilities is equivalent to applying the **softmax** transformation (see § 3.1) to the vector of dot-products,

$$\text{p}(\cdot \mid u) = \text{SoftMax}([\boldsymbol{\beta}_1 \cdot \boldsymbol{v}_u, \boldsymbol{\beta}_2 \cdot \boldsymbol{v}_u, \ldots, \boldsymbol{\beta}_V \cdot \boldsymbol{v}_u]). \quad [6.26]$$

The word vectors $\boldsymbol{\beta}_w$ are parameters of the model, and are estimated directly. The context vectors $\boldsymbol{v}_u$ can be computed in various ways, depending on the model. A simple but effective neural language model can be built from a **recurrent neural network** (RNN; Mikolov et al., 2010). The basic idea is to recurrently update the context vectors while moving through the sequence. Let $\boldsymbol{h}_m$ represent the contextual information at position m

[2] This idea predates neural language models (e.g., Rosenfeld, 1996; Roark et al., 2007).

in the sequence. RNN language models are defined,

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m} \quad [6.27]$$

$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \quad [6.28]$$

$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}, \quad [6.29]$$

where $\boldsymbol{\phi}$ is a matrix of **word embeddings**, and $\boldsymbol{x}_m$ denotes the embedding for word w_m. The conversion of w_m to $\boldsymbol{x}_m$ is sometimes known as a **lookup layer**, because we simply lookup the embeddings for each word in a table; see § 3.2.4.

The **Elman unit** defines a simple recurrent operation (Elman, 1990),

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta}\boldsymbol{h}_{m-1} + \boldsymbol{x}_m), \quad [6.30]$$

where $\boldsymbol{\Theta} \in \mathbb{R}^{K \times K}$ is the recurrence matrix and g is a non-linear transformation function, often defined as the elementwise hyperbolic tangent $\tanh$ (see § 3.1).[3] The $\tanh$ acts as a **squashing function**, ensuring that each element of $\boldsymbol{h}_m$ is constrained to the range $[-1, 1]$.

Although each w_m depends on only the context vector $\boldsymbol{h}_{m-1}$, this vector is in turn influenced by *all* previous tokens, $w_1, w_2, \ldots w_{m-1}$, through the recurrence operation: w_1 affects $\boldsymbol{h}_1$, which affects $\boldsymbol{h}_2$, and so on, until the information is propagated all the way to $\boldsymbol{h}_{m-1}$, and then on to w_m (see Figure 6.1). This is an important distinction from n-gram language models, where any information outside the n-word window is ignored. In principle, the RNN language model can handle long-range dependencies, such as number agreement over long spans of text — although it would be difficult to know where exactly in the vector $\boldsymbol{h}_m$ this information is represented. The main limitation is that information is attenuated by repeated application of the squashing function g. **Long short-term memories** (LSTMs), described below, are a variant of RNNs that address this issue, using memory cells to propagate information through the sequence without applying nonlinearities (Hochreiter and Schmidhuber, 1997).

The denominator in Equation 6.29 is a computational bottleneck, because it involves a sum over the entire vocabulary. One solution is to use a **hierarchical softmax** function, which computes the sum more efficiently by organizing the vocabulary into a tree (Mikolov et al., 2011). Another strategy is to optimize an alternative metric, such as **noise-contrastive estimation** (Gutmann and Hyvärinen, 2012), which learns by distinguishing observed instances from artificial instances generated from a noise distribution (Mnih and Teh, 2012). Both of these strategies are described in § 14.5.3.

[3] In the original Elman network, the sigmoid function was used in place of $\tanh$. For an illuminating mathematical discussion of the advantages and disadvantages of various nonlinearities in recurrent neural networks, see the lecture notes from Cho (2015).

6.3.1 Backpropagation through time

The recurrent neural network language model has the following parameters:

- $\boldsymbol{\phi}_i \in \mathbb{R}^K$, the "input" word vectors (these are sometimes called **word embeddings**, since each word is embedded in a K-dimensional space; see chapter 14);
- $\boldsymbol{\beta}_i \in \mathbb{R}^K$, the "output" word vectors;
- $\boldsymbol{\Theta} \in \mathbb{R}^{K \times K}$, the recurrence operator;
- $\boldsymbol{h}_0$, the initial state.

Each of these parameters can be estimated by formulating an objective function over the training corpus, $L(\boldsymbol{w})$, and then applying backpropagation to obtain gradients on the parameters from a minibatch of training examples (see § 3.3.1). Gradient-based updates can be computed from an online learning algorithm such as stochastic gradient descent (see § 2.6.2).

The application of backpropagation to recurrent neural networks is known as **backpropagation through time**, because the gradients on units at time m depend in turn on the gradients of units at earlier times $n < m$. Let ℓ_{m+1} represent the negative log-likelihood of word $m + 1$,

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m). \tag{6.31}$$

We require the gradient of this loss with respect to each parameter, such as $\theta_{k,k'}$, an individual element in the recurrence matrix Θ. Since the loss depends on the parameters only through $\boldsymbol{h}_m$, we can apply the chain rule of differentiation,

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}. \tag{6.32}$$

The vector $\boldsymbol{h}_m$ depends on $\boldsymbol{\Theta}$ in several ways. First, $\boldsymbol{h}_m$ is computed by multiplying $\boldsymbol{\Theta}$ by the previous state $\boldsymbol{h}_{m-1}$. But the previous state $\boldsymbol{h}_{m-1}$ also depends on $\boldsymbol{\Theta}$:

$$\boldsymbol{h}_m = g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \tag{6.33}$$

$$\frac{\partial h_{m,k}}{\partial \theta_{k,k'}} = g'(x_{m,k} + \boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-1})(h_{m-1,k'} + \boldsymbol{\theta}_k \cdot \frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}), \tag{6.34}$$

where g' is the local derivative of the nonlinear function g. The key point in this equation is that the derivative $\frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$ depends on $\frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}$, which will depend in turn on $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \theta_{k,k'}}$, and so on, until reaching the initial state $\boldsymbol{h}_0$.

Each derivative $\frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$ will be reused many times: it appears in backpropagation from the loss ℓ_m, but also in all subsequent losses $\ell_{n>m}$. Neural network toolkits such as Torch (Collobert et al., 2011) and DyNet (Neubig et al., 2017) compute the necessary

derivatives automatically, and cache them for future use. An important distinction from the feedforward neural networks considered in chapter 3 is that the size of the computation graph is not fixed, but varies with the length of the input. This poses difficulties for toolkits that are designed around static computation graphs, such as TensorFlow (Abadi et al., 2016).[4]

6.3.2 Hyperparameters

The RNN language model has several hyperparameters that must be tuned to ensure good performance. The model capacity is controlled by the size of the word and context vectors K, which play a role that is somewhat analogous to the size of the n-gram context. For datasets that are large with respect to the vocabulary (i.e., there is a large token-to-type ratio), we can afford to estimate a model with a large K, which enables more subtle distinctions between words and contexts. When the dataset is relatively small, then K must be smaller too, or else the model may "memorize" the training data, and fail to generalize. Unfortunately, this general advice has not yet been formalized into any concrete formula for choosing K, and trial-and-error is still necessary. Overfitting can also be prevented by **dropout**, which involves randomly setting some elements of the computation to zero (Srivastava et al., 2014), forcing the learner not to rely too much on any particular dimension of the word or context vectors. The dropout rate must also be tuned on development data.

6.3.3 Gated recurrent neural networks

In principle, recurrent neural networks can propagate information across infinitely long sequences. But in practice, repeated applications of the nonlinear recurrence function causes this information to be quickly attenuated. The same problem affects learning: backpropagation can lead to **vanishing gradients** that decay to zero, or **exploding gradients** that increase towards infinity (Bengio et al., 1994). The exploding gradient problem can be addressed by clipping gradients at some maximum value (Pascanu et al., 2013). The other issues must be addressed by altering the model itself.

The **long short-term memory** (LSTM; Hochreiter and Schmidhuber, 1997) is a popular variant of RNNs that is more robust to these problems. This model augments the hidden state $\boldsymbol{h}_m$ with a **memory cell** $\boldsymbol{c}_m$. The value of the memory cell at each time m is a gated sum of two quantities: its previous value $\boldsymbol{c}_{m-1}$, and an "update" $\tilde{c}_m$, which is computed from the current input $\boldsymbol{x}_m$ and the previous hidden state $\boldsymbol{h}_{m-1}$. The next state $\boldsymbol{h}_m$ is then computed from the memory cell. Because the memory cell is not passed through a nonlinear squashing function during the update, it is possible for information to propagate through the network over long distances.

[4] See `https://www.tensorflow.org/tutorials/recurrent` (retrieved Feb 8, 2018).

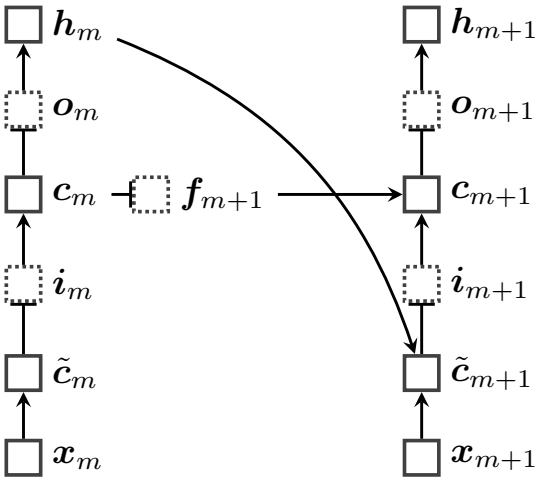


Figure 6.2: The long short-term memory (LSTM) architecture. Gates are shown in boxes with dotted edges. In an LSTM language model, each $\boldsymbol{h}_m$ would be used to predict the next word w_{m+1}.

The gates are functions of the input and previous hidden state. They are computed from elementwise sigmoid activations, $\sigma(x) = (1+\exp(-x))^{-1}$, ensuring that their values will be in the range $[0, 1]$. They can therefore be viewed as soft, differentiable logic gates. The LSTM architecture is shown in Figure 6.2, and the complete update equations are:

$$\boldsymbol{f}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\rightarrow f)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow f)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_f) \quad \text{forget gate} \quad [6.35]$$

$$\boldsymbol{i}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\rightarrow i)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow i)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_i) \quad \text{input gate} \quad [6.36]$$

$$\tilde{\boldsymbol{c}}_{m+1} = \tanh(\boldsymbol{\Theta}^{(h\rightarrow c)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(w\rightarrow c)}\boldsymbol{x}_{m+1}) \quad \text{update candidate} \quad [6.37]$$

$$\boldsymbol{c}_{m+1} = \boldsymbol{f}_{m+1} \odot \boldsymbol{c}_m + \boldsymbol{i}_{m+1} \odot \tilde{\boldsymbol{c}}_{m+1} \quad \text{memory cell update} \quad [6.38]$$

$$\boldsymbol{o}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\rightarrow o)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow o)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_o) \quad \text{output gate} \quad [6.39]$$

$$\boldsymbol{h}_{m+1} = \boldsymbol{o}_{m+1} \odot \tanh(\boldsymbol{c}_{m+1}) \quad \text{output.} \quad [6.40]$$

The operator $\odot$ is an elementwise (Hadamard) product. Each gate is controlled by a vector of weights, which parametrize the previous hidden state (e.g., $\boldsymbol{\Theta}^{(h\rightarrow f)}$) and the current input (e.g., $\boldsymbol{\Theta}^{(x\rightarrow f)}$), plus a vector offset (e.g., $\boldsymbol{b}_f$). The overall operation can be informally summarized as $(\boldsymbol{h}_m, \boldsymbol{c}_m) = \text{LSTM}(\boldsymbol{x}_m, (\boldsymbol{h}_{m-1}, \boldsymbol{c}_{m-1}))$, with $(\boldsymbol{h}_m, \boldsymbol{c}_m)$ representing the LSTM state after reading token m.

The LSTM outperforms standard recurrent neural networks across a wide range of problems. It was first used for language modeling by Sundermeyer et al. (2012), but can be applied more generally: the vector $\boldsymbol{h}_m$ can be treated as a complete representation of

the input sequence up to position m, and can be used for any labeling task on a sequence of tokens, as we will see in the next chapter.

There are several LSTM variants, of which the Gated Recurrent Unit (Cho et al., 2014) is one of the more well known. Many software packages implement a variety of RNN architectures, so choosing between them is simple from a user's perspective. Jozefowicz et al. (2015) provide an empirical comparison of various modeling choices circa 2015.

6.4 Evaluating language models

Language modeling is not usually an application in itself: language models are typically components of larger systems, and they would ideally be evaluated **extrinisically**. This means evaluating whether the language model improves performance on the application task, such as machine translation or speech recognition. But this is often hard to do, and depends on details of the overall system which may be irrelevant to language modeling. In contrast, **intrinsic evaluation** is task-neutral. Better performance on intrinsic metrics may be expected to improve extrinsic metrics across a variety of tasks, but there is always the risk of over-optimizing the intrinsic metric. This section discusses some intrinsic metrics, but keep in mind the importance of performing extrinsic evaluations to ensure that intrinsic performance gains carry over to real applications.

6.4.1 Held-out likelihood

The goal of probabilistic language models is to accurately measure the probability of sequences of word tokens. Therefore, an intrinsic evaluation metric is the likelihood that the language model assigns to **held-out data**, which is not used during training. Specifically, we compute,

$$\ell(\boldsymbol{w}) = \sum_{m=1}^{M} \log \mathrm{p}(w_m \mid w_{m-1}, \ldots, w_1), \tag{6.41}$$

treating the entire held-out corpus as a single stream of tokens.

Typically, unknown words are mapped to the $\langle \text{UNK} \rangle$ token. This means that we have to estimate some probability for $\langle \text{UNK} \rangle$ on the training data. One way to do this is to fix the vocabulary $\mathcal{V}$ to the $V - 1$ words with the highest counts in the training data, and then convert all other tokens to $\langle \text{UNK} \rangle$. Other strategies for dealing with out-of-vocabulary terms are discussed in § 6.5.

6.4.2 Perplexity

Held-out likelihood is usually presented as **perplexity**, which is a deterministic transformation of the log-likelihood into an information-theoretic quantity,

$$\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{\ell(\boldsymbol{w})}{M}}, \quad [6.42]$$

where M is the total number of tokens in the held-out corpus.

Lower perplexities correspond to higher likelihoods, so lower scores are better on this metric — it is better to be less perplexed. Here are some special cases:

- In the limit of a perfect language model, probability 1 is assigned to the held-out corpus, with $\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{1}{M}\log_2 1} = 2^0 = 1$.
- In the opposite limit, probability zero is assigned to the held-out corpus, which corresponds to an infinite perplexity, $\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{1}{M}\log_2 0} = 2^\infty = \infty$.
- Assume a uniform, unigram model in which $\text{p}(w_i) = \frac{1}{V}$ for all words in the vocabulary. Then,

$$\begin{aligned}
\log_2(\boldsymbol{w}) &= \sum_{m=1}^{M} \log_2 \frac{1}{V} = -\sum_{m=1}^{M} \log_2 V = -M \log_2 V \\
\text{Perplex}(\boldsymbol{w}) &= 2^{\frac{1}{M} M \log_2 V} \\
&= 2^{\log_2 V} \\
&= V.
\end{aligned}$$

  This is the "worst reasonable case" scenario, since you could build such a language model without even looking at the data.

In practice, language models tend to give perplexities in the range between 1 and V. A small benchmark dataset is the **Penn Treebank**, which contains roughly a million tokens; its vocabulary is limited to 10,000 words, with all other tokens mapped a special $\langle \text{UNK} \rangle$ symbol. On this dataset, a well-smoothed 5-gram model achieves a perplexity of 141 (Mikolov and Zweig, Mikolov and Zweig), and an LSTM language model achieves perplexity of roughly 80 (Zaremba, Sutskever, and Vinyals, Zaremba et al.). Various enhancements to the LSTM architecture can bring the perplexity below 60 (Merity et al., 2018). A larger-scale language modeling dataset is the 1B Word Benchmark (Chelba et al., 2013), which contains text from Wikipedia. On this dataset, perplexities of around 25 can be obtained by averaging together multiple LSTM language models (Jozefowicz et al., 2016).

6.5 Out-of-vocabulary words

So far, we have assumed a **closed-vocabulary** setting — the vocabulary $\mathcal{V}$ is assumed to be a finite set. In realistic application scenarios, this assumption may not hold. Consider, for example, the problem of translating newspaper articles. The following sentence appeared in a Reuters article on January 6, 2017:[5]

> The report said U.S. intelligence agencies believe Russian military intelligence, the **GRU**, used intermediaries such as **WikiLeaks**, **DCLeaks.com** and the **Guccifer** 2.0 "persona" to release emails...

Suppose that you trained a language model on the Gigaword corpus,[6] which was released in 2003. The bolded terms either did not exist at this date, or were not widely known; they are unlikely to be in the vocabulary. The same problem can occur for a variety of other terms: new technologies, previously unknown individuals, new words (e.g., *hashtag*), and numbers.

One solution is to simply mark all such terms with a special token, $\langle \text{UNK} \rangle$. While training the language model, we decide in advance on the vocabulary (often the K most common terms), and mark all other terms in the training data as $\langle \text{UNK} \rangle$. If we do not want to determine the vocabulary size in advance, an alternative approach is to simply mark the first occurrence of each word type as $\langle \text{UNK} \rangle$.

But is often better to make distinctions about the likelihood of various unknown words. This is particularly important in languages that have rich morphological systems, with many inflections for each word. For example, Portuguese is only moderately complex from a morphological perspective, yet each verb has dozens of inflected forms (see Figure 4.3b). In such languages, there will be many word types that we do not encounter in a corpus, which are nonetheless predictable from the morphological rules of the language. To use a somewhat contrived English example, if *transfenestrate* is in the vocabulary, our language model should assign a non-zero probability to the past tense *transfenestrated*, even if it does not appear in the training data.

One way to accomplish this is to supplement word-level language models with **character-level language models**. Such models can use n-grams or RNNs, but with a fixed vocabulary equal to the set of ASCII or Unicode characters. For example, Ling et al. (2015) propose an LSTM model over characters, and Kim (2014) employ a convolutional neural network. A more linguistically motivated approach is to segment words into meaningful subword units, known as **morphemes** (see chapter 9). For example, Botha and Blunsom

[5] Bayoumy, Y. and Strobel, W. (2017, January 6). U.S. intel report: Putin directed cyber campaign to help Trump. *Reuters*. Retrieved from `http://www.reuters.com/article/us-usa-russia-cyber-idUSKBN14Q1T8` on January 7, 2017.

[6] `https://catalog.ldc.upenn.edu/LDC2003T05`

(2014) induce vector representations for morphemes, which they build into a log-bilinear language model; Bhatia et al. (2016) incorporate morpheme vectors into an LSTM.

Additional resources

A variety of neural network architectures have been applied to language modeling. Notable earlier non-recurrent architectures include the neural probabilistic language model (Bengio et al., 2003) and the log-bilinear language model (Mnih and Hinton, 2007). Much more detail on these models can be found in the text by Goodfellow et al. (2016).

Exercises

1. Prove that n-gram language models give valid probabilities if the n-gram probabilities are valid. Specifically, assume that,

$$\sum_{w_m}^{\mathcal{V}} \mathrm{p}(w_m \mid w_{m-1}, w_{m-2}, \ldots, w_{m-n+1}) = 1 \quad [6.43]$$

   for all contexts $(w_{m-1}, w_{m-2}, \ldots, w_{m-n+1})$. Prove that $\sum_{\boldsymbol{w}} \mathrm{p}_n(\boldsymbol{w}) = 1$ for all $\boldsymbol{w} \in \mathcal{V}^*$, where p_n is the probability under an n-gram language model. Your proof should proceed by induction. You should handle the start-of-string case $\mathrm{p}(w_1 \mid \underbrace{\square, \ldots, \square}_{n-1})$, but you need not handle the end-of-string token.

2. First, show that RNN language models are valid using a similar proof technique to the one in the previous problem.

   Next, let $\mathrm{p}_r(\boldsymbol{w})$ indicate the probability of $\boldsymbol{w}$ under RNN r. An ensemble of RNN language models computes the probability,

$$\mathrm{p}(\boldsymbol{w}) = \frac{1}{R} \sum_{r=1}^{R} \mathrm{p}_r(\boldsymbol{w}). \quad [6.44]$$

   Does an ensemble of RNN language models compute a valid probability?

3. Consider a unigram language model over a vocabulary of size V. Suppose that a word appears m times in a corpus with M tokens in total. With Lidstone smoothing of α, for what values of m is the smoothed probability greater than the unsmoothed probability?

4. Consider a simple language in which each token is drawn from the vocabulary $\mathcal{V}$ with probability $\frac{1}{V}$, independent of all other tokens.

Given a corpus of size M, what is the expectation of the fraction of all possible bigrams that have zero count? You may assume V is large enough that $\frac{1}{V} \approx \frac{1}{V-1}$.

5. Continuing the previous problem, determine the value of M such that the fraction of bigrams with zero count is at most $\epsilon \in (0, 1)$. As a hint, you may use the approximation $\ln(1 + \alpha) \approx \alpha$ for $\alpha \approx 0$.

6. In real languages, words probabilities are neither uniform nor independent. Assume that word probabilities are independent but not uniform, so that in general $\text{p}(w) \neq \frac{1}{V}$. Prove that the expected fraction of unseen bigrams will be higher than in the IID case.

7. Consider a recurrent neural network with a single hidden unit and a sigmoid activation, $h_m = \sigma(\theta h_{m-1} + x_m)$. Prove that if $|\theta| < 1$, then the gradient $\frac{\partial h_m}{\partial h_{m-k}}$ goes to zero as $k \to \infty$.[7]

8. **Zipf's law** states that if the word types in a corpus are sorted by frequency, then the frequency of the word at rank r is proportional to r^{-s}, where s is a free parameter, usually around 1. (Another way to view Zipf's law is that a plot of log frequency against log rank will be linear.) Solve for s using the counts of the first and second most frequent words, c_1 and c_2.

9. Download the wikitext-2 dataset.[8] Read in the training data and compute word counts. Estimate the Zipf's law coefficient by,

$$\hat{s} = \exp\left(\frac{(\log \boldsymbol{r}) \cdot (\log \boldsymbol{c})}{||\log \boldsymbol{r}||_2^2}\right), \quad [6.45]$$

where $\boldsymbol{r} = [1, 2, 3, \ldots]$ is the vector of ranks of all words in the corpus, and $\boldsymbol{c} = [c_1, c_2, c_3, \ldots]$ is the vector of counts of all words in the corpus, sorted in descending order.

Make a log-log plot of the observed counts, and the expected counts according to Zipf's law. The sum $\sum_{r=1}^{\infty} r^s = \zeta(s)$ is the Riemann zeta function, available in python's `scipy` library as `scipy.special.zeta`.

10. Using the Pytorch library, train an LSTM language model from the Wikitext training corpus. After each epoch of training, compute its perplexity on the Wikitext validation corpus. Stop training when the perplexity stops improving.

[7] This proof generalizes to vector hidden units by considering the largest eigenvector of the matrix $\boldsymbol{\Theta}$ (Pascanu et al., 2013).

[8] Available at `https://github.com/pytorch/examples/tree/master/word_language_model/data/wikitext-2` in September 2018. The dataset is already tokenized, and already replaces rare words with ⟨UNK⟩, so no preprocessing is necessary.

Chapter 7

Sequence labeling

The goal of sequence labeling is to assign tags to words, or more generally, to assign discrete labels to discrete elements in a sequence. There are many applications of sequence labeling in natural language processing, and chapter 8 presents an overview. For now, we'll focus on the classic problem of **part-of-speech tagging**, which requires tagging each word by its grammatical category. Coarse-grained grammatical categories include **N**OUNs, which describe things, properties, or ideas, and **V**ERBs, which describe actions and events. Consider a simple input:

(7.1) They can fish.

A dictionary of coarse-grained part-of-speech tags might include NOUN as the only valid tag for *they*, but both NOUN and VERB as potential tags for *can* and *fish*. A accurate sequence labeling algorithm should select the verb tag for both *can* and *fish* in (7.1), but it should select noun for the same two words in the phrase *can of fish*.

7.1 Sequence labeling as classification

One way to solve a tagging problem is to turn it into a classification problem. Let $\boldsymbol{f}((\boldsymbol{w}, m), y)$ indicate the feature function for tag y at position m in the sequence $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$. A simple tagging model would have a single base feature, the word itself:

$$\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, m = 1), \text{N}) = (\textit{they}, \text{N}) \quad [7.1]$$
$$\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, m = 2), \text{V}) = (\textit{can}, \text{V}) \quad [7.2]$$
$$\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, m = 3), \text{V}) = (\textit{fish}, \text{V}). \quad [7.3]$$

Here the feature function takes three arguments as input: the sentence to be tagged (e.g., *they can fish*), the proposed tag (e.g., N or V), and the index of the token to which this tag

is applied. This simple feature function then returns a single feature: a tuple including the word to be tagged and the tag that has been proposed. If the vocabulary size is V and the number of tags is K, then there are $V \times K$ features. Each of these features must be assigned a weight. These weights can be learned from a labeled dataset using a classification algorithm such as perceptron, but this isn't necessary in this case: it would be equivalent to define the classification weights directly, with $\theta_{w,y} = 1$ for the tag y most frequently associated with word w, and $\theta_{w,y} = 0$ for all other tags.

However, it is easy to see that this simple classification approach cannot correctly tag both *they can fish* and *can of fish*, because *can* and *fish* are grammatically ambiguous. To handle both of these cases, the tagger must rely on context, such as the surrounding words. We can build context into the feature set by incorporating the surrounding words as additional features:

$$\begin{aligned}
\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, 1), \text{N}) = \{&(w_m = \textit{they}, y_m = \text{N}), \\
&(w_{m-1} = \square, y_m = \text{N}), \\
&(w_{m+1} = \textit{can}, y_m = \text{N})\} \qquad [7.4] \\
\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, 2), \text{V}) = \{&(w_m = \textit{can}, y_m = \text{V}), \\
&(w_{m-1} = \textit{they}, y_m = \text{V}), \\
&(w_{m+1} = \textit{fish}, y_m = \text{V})\} \qquad [7.5] \\
\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, 3), \text{V}) = \{&(w_m = \textit{fish}, y_m = \text{V}), \\
&(w_{m-1} = \textit{can}, y_m = \text{V}), \\
&(w_{m+1} = \blacksquare, y_m = \text{V})\}. \qquad [7.6]
\end{aligned}$$

These features contain enough information that a tagger should be able to choose the right tag for the word *fish*: words that come after *can* are likely to be verbs, so the feature $(w_{m-1} = \textit{can}, y_m = \text{V})$ should have a large positive weight.

However, even with this enhanced feature set, it may be difficult to tag some sequences correctly. One reason is that there are often relationships between the tags themselves. For example, in English it is relatively rare for a verb to follow another verb — particularly if we differentiate **M**ODAL verbs like *can* and *should* from more typical verbs, like *give*, *transcend*, and *befuddle*. We would like to incorporate preferences against tag sequences like VERB-VERB, and in favor of tag sequences like NOUN-VERB. The need for such preferences is best illustrated by a **garden path sentence**:

(7.2) The old man the boat.

Grammatically, the word *the* is a **D**ETERMINER. When you read the sentence, what part of speech did you first assign to *old*? Typically, this word is an ADJECTIVE — abbreviated as J — which is a class of words that modify nouns. Similarly, *man* is usually a noun. The resulting sequence of tags is D J N D N. But this is a mistaken "garden path" interpretation, which ends up leading nowhere. It is unlikely that a determiner would directly

follow a noun,[1] and it is particularly unlikely that the entire sentence would lack a verb. The only possible verb in (7.2) is the word *man*, which can refer to the act of maintaining and piloting something — often boats. But if *man* is tagged as a verb, then *old* is seated between a determiner and a verb, and must be a noun. And indeed, adjectives often have a second interpretation as nouns when used in this way (e.g., *the young*, *the restless*). This reasoning, in which the labeling decisions are intertwined, cannot be applied in a setting where each tag is produced by an independent classification decision.

7.2 Sequence labeling as structure prediction

As an alternative, think of the entire sequence of tags as a label itself. For a given sequence of words $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$, there is a set of possible taggings $\mathcal{Y}(\boldsymbol{w}) = \mathcal{Y}^M$, where $\mathcal{Y} = \{\text{N}, \text{V}, \text{D}, \ldots\}$ refers to the set of individual tags, and $\mathcal{Y}^M$ refers to the set of tag sequences of length M. We can then treat the sequence labeling problem as a classification problem in the label space $\mathcal{Y}(\boldsymbol{w})$,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}}\, \Psi(\boldsymbol{w}, \boldsymbol{y}), \tag{7.7}$$

where $\boldsymbol{y} = (y_1, y_2, \ldots, y_M)$ is a sequence of M tags, and Ψ is a scoring function on pairs of sequences, $V^M \times \mathcal{Y}^M \rightarrow \mathbb{R}$. Such a function can include features that capture the relationships between tagging decisions, such as the preference that determiners not follow nouns, or that all sentences have verbs.

Given that the label space is exponentially large in the length of the sequence M, can it ever be practical to perform tagging in this way? The problem of making a series of interconnected labeling decisions is known as **inference**. Because natural language is full of interrelated grammatical structures, inference is a crucial aspect of natural language processing. In English, it is not unusual to have sentences of length $M = 20$; part-of-speech tag sets vary in size from 10 to several hundred. Taking the low end of this range, we have $|\mathcal{Y}(\boldsymbol{w}_{1:M})| \approx 10^{20}$, one hundred billion billion possible tag sequences. Enumerating and scoring each of these sequences would require an amount of work that is exponential in the sequence length, so inference is intractable.

However, the situation changes when we restrict the scoring function. Suppose we choose a function that decomposes into a sum of local parts,

$$\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m), \tag{7.8}$$

where each $\psi(\cdot)$ scores a local part of the tag sequence. Note that the sum goes up to $M+1$, so that we can include a score for a special end-of-sequence tag, $\psi(\boldsymbol{w}_{1:M}, \blacklozenge, y_M, M+1)$. We also define a special tag to begin the sequence, $y_0 \triangleq \lozenge$.

[1] The main exception occurs with ditransitive verbs, such as *They gave the winner a trophy.*

In a linear model, local scoring function can be defined as a dot product of weights and features,

$$\psi(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \quad [7.9]$$

The feature vector $\boldsymbol{f}$ can consider the entire input $\boldsymbol{w}$, and can look at pairs of adjacent tags. This is a step up from per-token classification: the weights can assign low scores to infelicitous tag pairs, such as noun-determiner, and high scores for frequent tag pairs, such as determiner-noun and noun-verb.

In the example *they can fish*, a minimal feature function would include features for word-tag pairs (sometimes called **emission features**) and tag-tag pairs (sometimes called **transition features**):

$$\begin{aligned}
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \boldsymbol{y} = \text{N V V }) &= \sum_{m=1}^{M+1} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) && [7.10] \\
&= \boldsymbol{f}(\boldsymbol{w}, \text{N}, \lozenge, 1) \\
&\quad + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{N}, 2) \\
&\quad + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{V}, 3) \\
&\quad + \boldsymbol{f}(\boldsymbol{w}, \blacklozenge, \text{V}, 4) && [7.11] \\
&= (w_m = \textit{they}, y_m = \text{N}) + (y_m = \text{N}, y_{m-1} = \lozenge) \\
&\quad + (w_m = \textit{can}, y_m = \text{V}) + (y_m = \text{V}, y_{m-1} = \text{N}) \\
&\quad + (w_m = \textit{fish}, y_m = \text{V}) + (y_m = \text{V}, y_{m-1} = \text{V}) \\
&\quad + (y_m = \blacklozenge, y_{m-1} = \text{V}). && [7.12]
\end{aligned}$$

There are seven active features for this example: one for each word-tag pair, and one for each tag-tag pair, including a final tag $y_{M+1} = \blacklozenge$. These features capture the two main sources of information for part-of-speech tagging in English: which tags are appropriate for each word, and which tags tend to follow each other in sequence. Given appropriate weights for these features, taggers can achieve high accuracy, even for difficult cases like *the old man the boat*. We will now discuss how this restricted scoring function enables efficient inference, through the **Viterbi algorithm** (Viterbi, 1967).

7.3 The Viterbi algorithm

By decomposing the scoring function into a sum of local parts, it is possible to rewrite the tagging problem as follows:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}} \Psi(\boldsymbol{w}, \boldsymbol{y}) \tag{7.13}$$

$$= \underset{\boldsymbol{y}_{1:M}}{\text{argmax}} \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{7.14}$$

$$= \underset{\boldsymbol{y}_{1:M}}{\text{argmax}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}), \tag{7.15}$$

where the final line simplifies the notation with the shorthand,

$$s_m(y_m, y_{m-1}) \triangleq \psi(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m). \tag{7.16}$$

This inference problem can be solved efficiently using **dynamic programming**, an algorithmic technique for reusing work in recurrent computations. We begin by solving an auxiliary problem: rather than finding the best tag sequence, we compute the *score* of the best tag sequence,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}, \boldsymbol{y}_{1:M}) = \max_{\boldsymbol{y}_{1:M}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}). \tag{7.17}$$

This score involves a maximization over all tag sequences of length M, written $\max_{\boldsymbol{y}_{1:M}}$. This maximization can be broken into two pieces,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}, \boldsymbol{y}_{1:M}) = \max_{y_M} \max_{\boldsymbol{y}_{1:M-1}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}). \tag{7.18}$$

Within the sum, only the final term $s_{M+1}(\blacklozenge, y_M)$ depends on y_M, so we can pull this term out of the second maximization,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}, \boldsymbol{y}_{1:M}) = \left(\max_{y_M} s_{M+1}(\blacklozenge, y_M) \right) + \left(\max_{\boldsymbol{y}_{1:M-1}} \sum_{m=1}^{M} s_m(y_m, y_{m-1}) \right). \tag{7.19}$$

The second term in Equation 7.19 has the same form as our original problem, with M replaced by $M-1$. This indicates that the problem can be reformulated as a recurrence. We do this by defining an auxiliary variable called the **Viterbi variable** $v_m(k)$, representing

Algorithm 11 The Viterbi algorithm. Each $s_m(k, k')$ is a local score for tag $y_m = k$ and $y_{m-1} = k'$.

for $k \in \{0, \dots K\}$ **do**
  $v_1(k) = s_1(k, \lozenge)$
for $m \in \{2, \dots, M\}$ **do**
  for $k \in \{0, \dots, K\}$ **do**
    $v_m(k) = \max_{k'} s_m(k, k') + v_{m-1}(k')$
    $b_m(k) = \text{argmax}_{k'} s_m(k, k') + v_{m-1}(k')$
$y_M = \text{argmax}_k s_{M+1}(\blacklozenge, k) + v_M(k)$
for $m \in \{M-1, \dots 1\}$ **do**
  $y_m = b_m(y_{m+1})$
return $\boldsymbol{y}_{1:M}$

the score of the best sequence terminating in the tag k:

$$v_m(y_m) \triangleq \max_{\boldsymbol{y}_{1:m-1}} \sum_{n=1}^{m} s_n(y_n, y_{n-1}) \tag{7.20}$$

$$= \max_{y_{m-1}} s_m(y_m, y_{m-1}) + \max_{\boldsymbol{y}_{1:m-2}} \sum_{n=1}^{m-1} s_n(y_n, y_{n-1}) \tag{7.21}$$

$$= \max_{y_{m-1}} s_m(y_m, y_{m-1}) + v_{m-1}(y_{m-1}). \tag{7.22}$$

Each set of Viterbi variables is computed from the local score $s_m(y_m, y_{m-1})$, and from the previous set of Viterbi variables. The initial condition of the recurrence is simply the score for the first tag,

$$v_1(y_1) \triangleq s_1(y_1, \lozenge). \tag{7.23}$$

The maximum overall score for the sequence is then the final Viterbi variable,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}_{1:M}, \boldsymbol{y}_{1:M}) = v_{M+1}(\blacklozenge). \tag{7.24}$$

Thus, the score of the best labeling for the sequence can be computed in a single forward sweep: first compute all variables $v_1(\cdot)$ from Equation 7.23, and then compute all variables $v_2(\cdot)$ from the recurrence in Equation 7.22, continuing until the final variable $v_{M+1}(\blacklozenge)$.

The Viterbi variables can be arranged in a structure known as a **trellis**, shown in Figure 7.1. Each column indexes a token m in the sequence, and each row indexes a tag in $\mathcal{Y}$; every $v_{m-1}(k)$ is connected to every $v_m(k')$, indicating that $v_m(k')$ is computed from $v_{m-1}(k)$. Special nodes are set aside for the start and end states.

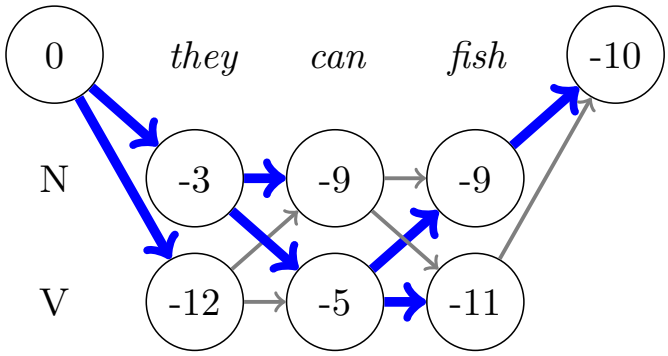


Figure 7.1: The trellis representation of the Viterbi variables, for the example *they can fish*, using the weights shown in Table 7.1.

The original goal was to find the best scoring sequence, not simply to compute its score. But by solving the auxiliary problem, we are almost there. Recall that each $v_m(k)$ represents the score of the best tag sequence ending in that tag k in position m. To compute this, we maximize over possible values of y_{m-1}. By keeping track of the "argmax" tag that maximizes this choice at each step, we can walk backwards from the final tag, and recover the optimal tag sequence. This is indicated in Figure 7.1 by the thick lines, which we trace back from the final position. These backward pointers are written $b_m(k)$, indicating the optimal tag y_{m-1} on the path to $Y_m = k$.

The complete Viterbi algorithm is shown in Algorithm 11. When computing the initial Viterbi variables $v_1(\cdot)$, the special tag $\lozenge$ indicates the start of the sequence. When computing the final tag Y_M, another special tag,$\blacklozenge$ indicates the end of the sequence. These special tags enable the use of transition features for the tags that begin and end the sequence: for example, conjunctions are unlikely to end sentences in English, so we would like a low score for $s_{M+1}(\blacklozenge, \text{CC})$; nouns are relatively likely to appear at the beginning of sentences, so we would like a high score for $s_1(\text{N}, \lozenge)$, assuming the noun tag is compatible with the first word token w_1.

Complexity If there are K tags and M positions in the sequence, then there are $M \times K$ Viterbi variables to compute. Computing each variable requires finding a maximum over K possible predecessor tags. The total time complexity of populating the trellis is therefore $\mathcal{O}(MK^2)$, with an additional factor for the number of active features at each position. After completing the trellis, we simply trace the backwards pointers to the beginning of the sequence, which takes $\mathcal{O}(M)$ operations.

	they	*can*	*fish*
N	-2	-3	-3
V	-10	-1	-3

(a) Weights for emission features.

	N	V	⧫
◊	-1	-2	$-\infty$
N	-3	-1	-1
V	-1	-3	-1

(b) Weights for transition features. The "from" tags are on the columns, and the "to" tags are on the rows.

Table 7.1: Feature weights for the example trellis shown in Figure 7.1. Emission weights from ◊ and ⧫ are implicitly set to $-\infty$.

7.3.1 Example

Consider the minimal tagset $\{\text{N}, \text{V}\}$, corresponding to nouns and verbs. Even in this tagset, there is considerable ambiguity: for example, the words *can* and *fish* can each take both tags. Of the $2 \times 2 \times 2 = 8$ possible taggings for the sentence *they can fish*, four are possible given these possible tags, and two are grammatical.[2]

The values in the trellis in Figure 7.1 are computed from the feature weights defined in Table 7.1. We begin with $v_1(\text{N})$, which has only one possible predecessor, the start tag ◊. This score is therefore equal to $s_1(\text{N}, ◊) = -2 - 1 = -3$, which is the sum of the scores for the emission and transition features respectively; the backpointer is $b_1(\text{N}) = ◊$. The score for $v_1(\text{V})$ is computed in the same way: $s_1(\text{V}, ◊) = -10 - 2 = -12$, and again $b_1(\text{V}) = ◊$. The backpointers are represented in the figure by thick lines.

Things get more interesting at $m = 2$. The score $v_2(\text{N})$ is computed by maximizing over the two possible predecessors,

$$\begin{aligned} v_2(\text{N}) &= \max(v_1(\text{N}) + s_2(\text{N}, \text{N}), v_1(\text{V}) + s_2(\text{N}, \text{V})) && [7.25] \\ &= \max(-3 - 3 - 3, \quad -12 - 3 - 1) = -9 && [7.26] \\ b_2(\text{N}) &= \text{N}. && [7.27] \end{aligned}$$

This continues until reaching $v_4(⧫)$, which is computed as,

$$\begin{aligned} v_4(⧫) &= \max(v_3(\text{N}) + s_4(⧫, \text{N}), v_3(\text{V}) + s_4(⧫, \text{V})) && [7.28] \\ &= \max(-9 + 0 - 1, \quad -11 + 0 - 1) && [7.29] \\ &= -10, && [7.30] \end{aligned}$$

so $b_4(⧫) = \text{N}$. As there is no emission w_4, the emission features have scores of zero.

[2] The tagging *they*/N *can*/V *fish*/N corresponds to the scenario of putting fish into cans, or perhaps of firing them.

To compute the optimal tag sequence, we walk backwards from here, next checking $b_3(\text{N}) = \text{V}$, and then $b_2(\text{V}) = \text{N}$, and finally $b_1(\text{N}) = \lozenge$. This yields $\boldsymbol{y} = (\text{N}, \text{V}, \text{N})$, which corresponds to the linguistic interpretation of the fishes being put into cans.

7.3.2 Higher-order features

The Viterbi algorithm was made possible by a restriction of the scoring function to local parts that consider only pairs of adjacent tags. We can think of this as a bigram language model over tags. A natural question is how to generalize Viterbi to tag trigrams, which would involve the following decomposition:

$$\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+2} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, y_{m-2}, m), \quad [7.31]$$

where $y_{-1} = \lozenge$ and $y_{M+2} = \blacklozenge$.

One solution is to create a new tagset $\mathcal{Y}^{(2)}$ from the Cartesian product of the original tagset with itself, $\mathcal{Y}^{(2)} = \mathcal{Y} \times \mathcal{Y}$. The tags in this product space are ordered pairs, representing adjacent tags at the token level: for example, the tag (N, V) would represent a noun followed by a verb. Transitions between such tags must be consistent: we can have a transition from (N, V) to (V, N) (corresponding to the tag sequence N V N), but not from (N, V) to (N, N), which would not correspond to any coherent tag sequence. This constraint can be enforced in feature weights, with $\theta_{((a,b),(c,d))} = -\infty$ if $b \neq c$. The remaining feature weights can encode preferences for and against various tag trigrams.

In the Cartesian product tag space, there are K^2 tags, suggesting that the time complexity will increase to $\mathcal{O}(MK^4)$. However, it is unnecessary to $\max$ over predecessor tag bigrams that are incompatible with the current tag bigram. By exploiting this constraint, it is possible to limit the time complexity to $\mathcal{O}(MK^3)$. The space complexity grows to $\mathcal{O}(MK^2)$, since the trellis must store all possible predecessors of each tag. In general, the time and space complexity of higher-order Viterbi grows exponentially with the order of the tag n-grams that are considered in the feature decomposition.

7.4 Hidden Markov Models

The Viterbi sequence labeling algorithm is built on the scores $s_m(y, y')$. We will now discuss how these scores can be estimated probabilistically. Recall from § 2.2 that the probabilistic Naïve Bayes classifier selects the label y to maximize $\text{p}(y \mid \boldsymbol{x}) \propto \text{p}(y, \boldsymbol{x})$. In probabilistic sequence labeling, our goal is similar: select the tag sequence that maximizes $\text{p}(\boldsymbol{y} \mid \boldsymbol{w}) \propto \text{p}(\boldsymbol{y}, \boldsymbol{w})$. The locality restriction in Equation 7.8 can be viewed as a conditional independence assumption on the random variables $\boldsymbol{y}$.

Algorithm 12 Generative process for the hidden Markov model

$y_0 \leftarrow \lozenge, \quad m \leftarrow 1$
repeat
  $y_m \sim \text{Categorical}(\boldsymbol{\lambda}_{y_{m-1}})$ ▷ sample the current tag
  $w_m \sim \text{Categorical}(\boldsymbol{\phi}_{y_m})$ ▷ sample the current word
until $y_m = \blacklozenge$ ▷ terminate when the stop symbol is generated

Naïve Bayes was introduced as a **generative model** — a probabilistic story that explains the observed data as well as the hidden label. A similar story can be constructed for probabilistic sequence labeling: first, the tags are drawn from a prior distribution; next, the tokens are drawn from a conditional likelihood. However, for inference to be tractable, additional independence assumptions are required. First, the probability of each token depends only on its tag, and not on any other element in the sequence:

$$\text{p}(\boldsymbol{w} \mid \boldsymbol{y}) = \prod_{m=1}^{M} \text{p}(w_m \mid y_m). \quad [7.32]$$

Second, each tag y_m depends only on its predecessor,

$$\text{p}(\boldsymbol{y}) = \prod_{m=1}^{M} \text{p}(y_m \mid y_{m-1}), \quad [7.33]$$

where $y_0 = \lozenge$ in all cases. Due to this **Markov assumption**, probabilistic sequence labeling models are known as **hidden Markov models** (HMMs).

The generative process for the hidden Markov model is shown in Algorithm 12. Given the parameters $\boldsymbol{\lambda}$ and $\boldsymbol{\phi}$, we can compute $\text{p}(\boldsymbol{w}, \boldsymbol{y})$ for any token sequence $\boldsymbol{w}$ and tag sequence $\boldsymbol{y}$. The HMM is often represented as a **graphical model** (Wainwright and Jordan, 2008), as shown in Figure 7.2. This representation makes the independence assumptions explicit: if a variable v_1 is probabilistically conditioned on another variable v_2, then there is an arrow $v_2 \rightarrow v_1$ in the diagram. If there are no arrows between v_1 and v_2, they are **conditionally independent**, given each variable's **Markov blanket**. In the hidden Markov model, the Markov blanket for each tag y_m includes the "parent" y_{m-1}, and the "children" y_{m+1} and w_m.[3]

It is important to reflect on the implications of the HMM independence assumptions. A non-adjacent pair of tags y_m and y_n are conditionally independent; if $m < n$ and we are given y_{n-1}, then y_m offers no additional information about y_n. However, if we are not given any information about the tags in a sequence, then all tags are probabilistically coupled.

[3] In general graphical models, a variable's Markov blanket includes its parents, children, and its children's other parents (Murphy, 2012).

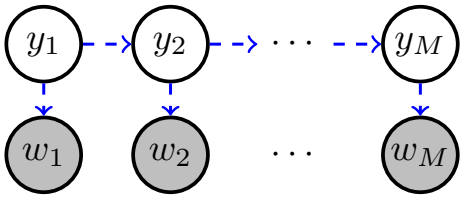


Figure 7.2: Graphical representation of the hidden Markov model. Arrows indicate probabilistic dependencies.

7.4.1 Estimation

The hidden Markov model has two groups of parameters:

Emission probabilities. The probability $\text{p}_e(w_m \mid y_m; \phi)$ is the emission probability, since the words are treated as probabilistically "emitted", conditioned on the tags.

Transition probabilities. The probability $\text{p}_t(y_m \mid y_{m-1}; \lambda)$ is the transition probability, since it assigns probability to each possible tag-to-tag transition.

Both of these groups of parameters are typically computed from smoothed relative frequency estimation on a labeled corpus (see § 6.2 for a review of smoothing). The unsmoothed probabilities are,

$$\phi_{k,i} \triangleq \Pr(W_m = i \mid Y_m = k) = \frac{\text{count}(W_m = i, Y_m = k)}{\text{count}(Y_m = k)}$$
$$\lambda_{k,k'} \triangleq \Pr(Y_m = k' \mid Y_{m-1} = k) = \frac{\text{count}(Y_m = k', Y_{m-1} = k)}{\text{count}(Y_{m-1} = k)}.$$

Smoothing is more important for the emission probability than the transition probability, because the vocabulary is much larger than the number of tags.

7.4.2 Inference

The goal of inference in the hidden Markov model is to find the highest probability tag sequence,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\text{argmax}}\, \text{p}(\boldsymbol{y} \mid \boldsymbol{w}). \quad [7.34]$$

As in Naïve Bayes, it is equivalent to find the tag sequence with the highest *log*-probability, since the logarithm is a monotonically increasing function. It is furthermore equivalent to maximize the joint probability $\text{p}(\boldsymbol{y}, \boldsymbol{w}) = \text{p}(\boldsymbol{y} \mid \boldsymbol{w}) \times \text{p}(\boldsymbol{w}) \propto \text{p}(\boldsymbol{y} \mid \boldsymbol{w})$, which is proportional to the conditional probability. Putting these observations together, the inference

problem can be reformulated as,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}). \tag{7.35}$$

We can now apply the HMM independence assumptions:

$$\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \log \mathrm{p}(\boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) \tag{7.36}$$

$$= \sum_{m=1}^{M+1} \log \mathrm{p}_Y(y_m \mid y_{m-1}) + \log \mathrm{p}_{W|Y}(w_m \mid y_m) \tag{7.37}$$

$$= \sum_{m=1}^{M+1} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m} \tag{7.38}$$

$$= \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}), \tag{7.39}$$

where,

$$s_m(y_m, y_{m-1}) \triangleq \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m}, \tag{7.40}$$

and,

$$\phi_{\blacklozenge, w} = \begin{cases} 1, & w = \blacksquare \\ 0, & \text{otherwise}, \end{cases} \tag{7.41}$$

which ensures that the stop tag $\blacklozenge$ can only be applied to the final token $\blacksquare$.

This derivation shows that HMM inference can be viewed as an application of the Viterbi decoding algorithm, given an appropriately defined scoring function. The local score $s_m(y_m, y_{m-1})$ can be interpreted probabilistically,

$$s_m(y_m, y_{m-1}) = \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) \tag{7.42}$$

$$= \log \mathrm{p}(y_m, w_m \mid y_{m-1}). \tag{7.43}$$

Now recall the definition of the Viterbi variables,

$$v_m(y_m) = \max_{y_{m-1}} s_m(y_m, y_{m-1}) + v_{m-1}(y_{m-1}) \tag{7.44}$$

$$= \max_{y_{m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + v_{m-1}(y_{m-1}). \tag{7.45}$$

By setting $v_{m-1}(y_{m-1}) = \max_{\boldsymbol{y}_{1:m-2}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, \boldsymbol{w}_{1:m-1})$, we obtain the recurrence,

$$v_m(y_m) = \max_{y_{m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + \max_{\boldsymbol{y}_{1:m-2}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, \boldsymbol{w}_{1:m-1}) \tag{7.46}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, \boldsymbol{w}_{1:m-1}) \tag{7.47}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(\boldsymbol{y}_{1:m}, \boldsymbol{w}_{1:m}). \tag{7.48}$$

In words, the Viterbi variable $v_m(y_m)$ is the log probability of the best tag sequence ending in y_m, joint with the word sequence $\boldsymbol{w}_{1:m}$. The log probability of the best complete tag sequence is therefore,

$$\max_{\boldsymbol{y}_{1:M}} \log \text{p}(\boldsymbol{y}_{1:M+1}, \boldsymbol{w}_{1:M+1}) = v_{M+1}(\blacklozenge) \tag{7.49}$$

***Viterbi as an example of the max-product algorithm** The Viterbi algorithm can also be implemented using probabilities, rather than log-probabilities. In this case, each $v_m(y_m)$ is equal to,

$$\begin{aligned}
v_m(y_m) &= \max_{\boldsymbol{y}_{1:m-1}} \text{p}(\boldsymbol{y}_{1:m-1}, y_m, \boldsymbol{w}_{1:m}) && [7.50]\\
&= \max_{y_{m-1}} \text{p}(y_m, w_m \mid y_{m-1}) \times \max_{\boldsymbol{y}_{1:m-2}} \text{p}(y_{1:m-2}, y_{m-1}, \boldsymbol{w}_{1:m-1}) && [7.51]\\
&= \max_{y_{m-1}} \text{p}(y_m, w_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && [7.52]\\
&= \text{p}_{w|y}(w_m \mid y_m) \times \max_{y_{m-1}} \text{p}_y(y_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}). && [7.53]
\end{aligned}$$

Each Viterbi variable is computed by *maximizing* over a set of *products*. Thus, the Viterbi algorithm is a special case of the **max-product algorithm** for inference in graphical models (Wainwright and Jordan, 2008). However, the product of probabilities tends towards zero over long sequences, so the log-probability version of Viterbi is recommended in practical implementations.

7.5 Discriminative sequence labeling with features

Today, hidden Markov models are rarely used for supervised sequence labeling. This is because HMMs are limited to only two phenomena:

- word-tag compatibility, via the emission probability $\text{p}_{W|Y}(w_m \mid y_m)$;
- local context, via the transition probability $\text{p}_Y(y_m \mid y_{m-1})$.

The Viterbi algorithm permits the inclusion of richer information in the local scoring function $\psi(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m)$, which can be defined as a weighted sum of arbitrary local *features*,

$$\psi(\boldsymbol{w}, y_m, y_{m-1}, m) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m), \tag{7.54}$$

where $\boldsymbol{f}$ is a locally-defined feature function, and $\boldsymbol{\theta}$ is a vector of weights.

The local decomposition of the scoring function Ψ is reflected in a corresponding decomposition of the feature function:

$$\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m) \quad [7.55]$$

$$= \sum_{m=1}^{M+1} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \quad [7.56]$$

$$= \boldsymbol{\theta} \cdot \sum_{m=1}^{M+1} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \quad [7.57]$$

$$= \boldsymbol{\theta} \cdot \boldsymbol{f}^{(\text{global})}(\boldsymbol{w}, \boldsymbol{y}_{1:M}), \quad [7.58]$$

where $\boldsymbol{f}^{(\text{global})}(\boldsymbol{w}, \boldsymbol{y})$ is a global feature vector, which is a sum of local feature vectors,

$$\boldsymbol{f}^{(\text{global})}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \boldsymbol{f}(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m), \quad [7.59]$$

with $y_{M+1} = \blacklozenge$ and $y_0 = \lozenge$ by construction.

Let's now consider what additional information these features might encode.

Word affix features. Consider the problem of part-of-speech tagging on the first four lines of the poem *Jabberwocky* (Carroll, 1917):

(7.3) 'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

Many of these words were made up by the author of the poem, so a corpus would offer no information about their probabilities of being associated with any particular part of speech. Yet it is not so hard to see what their grammatical roles might be in this passage. Context helps: for example, the word *slithy* follows the determiner *the*, so it is probably a noun or adjective. Which do you think is more likely? The suffix *-thy* is found in a number of adjectives, like *frothy, healthy, pithy, worthy*. It is also found in a handful of nouns — e.g., *apathy, sympathy* — but nearly all of these have the longer coda *-pathy*, unlike *slithy*. So the suffix gives some evidence that *slithy* is an adjective, and indeed it is: later in the text we find that it is a combination of the adjectives *lithe* and *slimy*.[4]

[4] **Morphology** is the study of how words are formed from smaller linguistic units. chapter 9 touches on computational approaches to morphological analysis. See Bender (2013) for an overview of the underlying linguistic principles, and Haspelmath and Sims (2013) or Lieber (2015) for a full treatment.

Fine-grained context. The hidden Markov model captures contextual information in the form of part-of-speech tag bigrams. But sometimes, the necessary contextual information is more specific. Consider the noun phrases *this fish* and *these fish*. Many part-of-speech tagsets distinguish between singular and plural nouns, but do not distinguish between singular and plural determiners; for example, the well known **Penn Treebank** tagset follows these conventions. A hidden Markov model would be unable to correctly label *fish* as singular or plural in both of these cases, because it only has access to two features: the preceding tag (determiner in both cases) and the word (*fish* in both cases). The classification-based tagger discussed in § 7.1 had the ability to use preceding and succeeding words as features, and it can also be incorporated into a Viterbi-based sequence labeler as a local feature.

Example. Consider the tagging D J N (determiner, adjective, noun) for the sequence *the slithy toves*, so that

$$\begin{aligned}\boldsymbol{w} &= \textit{the slithy toves} \\ \boldsymbol{y} &= \text{D J N}.\end{aligned}$$

Let's create the feature vector for this example, assuming that we have word-tag features (indicated by W), tag-tag features (indicated by T), and suffix features (indicated by M). You can assume that you have access to a method for extracting the suffix *-thy* from *slithy*, *-es* from *toves*, and $\varnothing$ from *the*, indicating that this word has no suffix.[5] The resulting feature vector is,

$$\begin{aligned}\boldsymbol{f}(\text{the slithy toves}, \text{D J N}) =& \boldsymbol{f}(\text{the slithy toves}, \text{D}, \lozenge, 1) \\ &+ \boldsymbol{f}(\text{the slithy toves}, \text{J}, \text{D}, 2) \\ &+ \boldsymbol{f}(\text{the slithy toves}, \text{N}, \text{J}, 3) \\ &+ \boldsymbol{f}(\text{the slithy toves}, \blacklozenge, \text{N}, 4) \\ =& \{(T : \lozenge, \text{D}), (W : \text{the}, \text{D}), (M : \varnothing, \text{D}), \\ & (T : \text{D}, \text{J}), (W : \text{slithy}, \text{J}), (M : \text{-thy}, \text{J}), \\ & (T : \text{J}, \text{N}), (W : \text{toves}, \text{N}), (M : \text{-es}, \text{N}) \\ & (T : \text{N}, \blacklozenge)\}.\end{aligned}$$

These examples show that local features can incorporate information that lies beyond the scope of a hidden Markov model. Because the features are local, it is possible to apply the Viterbi algorithm to identify the optimal sequence of tags. The remaining question

[5] Such a system is called a **morphological segmenter**. The task of morphological segmentation is briefly described in § 9.1.4; a well known segmenter is MORFESSOR (Creutz and Lagus, 2007). In real applications, a typical approach is to include features for all orthographic suffixes up to some maximum number of characters: for *slithy*, we would have suffix features for *-y*, *-hy*, and *-thy*.

is how to estimate the weights on these features. § 2.3 presented three main types of discriminative classifiers: perceptron, support vector machine, and logistic regression. Each of these classifiers has a structured equivalent, enabling it to be trained from labeled sequences rather than individual tokens.

7.5.1 Structured perceptron

The perceptron classifier is trained by increasing the weights for features that are associated with the correct label, and decreasing the weights for features that are associated with incorrectly predicted labels:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) \quad [7.60]$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{f}(\boldsymbol{x}, \hat{y}). \quad [7.61]$$

We can apply exactly the same update in the case of structure prediction,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \quad [7.62]$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}). \quad [7.63]$$

This learning algorithm is called **structured perceptron**, because it learns to predict the structured output $\boldsymbol{y}$. The only difference is that instead of computing $\hat{\boldsymbol{y}}$ by enumerating the entire set $\mathcal{Y}$, the Viterbi algorithm is used to efficiently search the set of possible taggings, $\mathcal{Y}^M$. Structured perceptron can be applied to other structured outputs as long as efficient inference is possible. As in perceptron classification, weight averaging is crucial to get good performance (see § 2.3.2).

Example For the example *they can fish*, suppose that the reference tag sequence is $\boldsymbol{y}^{(i)} =$ N V V, but the tagger incorrectly returns the tag sequence $\hat{\boldsymbol{y}} =$ N V N. Assuming a model with features for emissions (w_m, y_m) and transitions (y_{m-1}, y_m), the corresponding structured perceptron update is:

$$\theta_{(\textit{fish},\text{V})} \leftarrow \theta_{(\textit{fish},\text{V})} + 1, \qquad \theta_{(\textit{fish},\text{N})} \leftarrow \theta_{(\textit{fish},\text{N})} - 1 \quad [7.64]$$

$$\theta_{(\text{V},\text{V})} \leftarrow \theta_{(\text{V},\text{V})} + 1, \qquad \theta_{(\text{V},\text{N})} \leftarrow \theta_{(\text{V},\text{N})} - 1 \quad [7.65]$$

$$\theta_{(\text{V},\blacklozenge)} \leftarrow \theta_{(\text{V},\blacklozenge)} + 1, \qquad \theta_{(\text{N},\blacklozenge)} \leftarrow \theta_{(\text{N},\blacklozenge)} - 1. \quad [7.66]$$

7.5.2 Structured support vector machines

Large-margin classifiers such as the support vector machine improve on the perceptron by pushing the classification boundary away from the training instances. The same idea can

be applied to sequence labeling. A support vector machine in which the output is a structured object, such as a sequence, is called a **structured support vector machine** (Tsochantaridis et al., 2004).[6]

In classification, we formalized the large-margin constraint as,

$$\forall y \neq y^{(i)}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) \geq 1, \tag{7.67}$$

requiring a margin of at least 1 between the scores for all labels y that are not equal to the correct label $y^{(i)}$. The weights $\boldsymbol{\theta}$ are then learned by constrained optimization (see § 2.4.2).

This idea can be applied to sequence labeling by formulating an equivalent set of constraints for all possible labelings $\mathcal{Y}(\boldsymbol{w})$ for an input $\boldsymbol{w}$. However, there are two problems. First, in sequence labeling, some predictions are more wrong than others: we may miss only one tag out of fifty, or we may get all fifty wrong. We would like our learning algorithm to be sensitive to this difference. Second, the number of constraints is equal to the number of possible labelings, which is exponentially large in the length of the sequence.

The first problem can be addressed by adjusting the constraint to require larger margins for more serious errors. Let $c(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}) \geq 0$ represent the *cost* of predicting label $\hat{\boldsymbol{y}}$ when the true label is $\boldsymbol{y}^{(i)}$. We can then generalize the margin constraint,

$$\forall \boldsymbol{y}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \geq c(\boldsymbol{y}^{(i)}, \boldsymbol{y}). \tag{7.68}$$

This cost-augmented margin constraint specializes to the constraint in Equation 7.67 if we choose the delta function $c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \delta\left(() \boldsymbol{y}^{(i)} \neq \boldsymbol{y}\right)$. A more expressive cost function is the **Hamming cost**,

$$c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \sum_{m=1}^{M} \delta(y_m^{(i)} \neq y_m), \tag{7.69}$$

which computes the number of errors in $\boldsymbol{y}$. By incorporating the cost function as the margin constraint, we require that the true labeling be seperated from the alternatives by a margin that is proportional to the number of incorrect tags in each alternative labeling.

The second problem is that the number of constraints is exponential in the length of the sequence. This can be addressed by focusing on the prediction $\hat{\boldsymbol{y}}$ that *maximally* violates the margin constraint. This prediction can be identified by solving the following **cost-augmented decoding** problem:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \tag{7.70}$$

$$= \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}), \tag{7.71}$$

[6] This model is also known as a **max-margin Markov network** (Taskar et al., 2003), emphasizing that the scoring function is constructed from a sum of components, which are Markov independent.

where in the second line we drop the term $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$, which is constant in $\boldsymbol{y}$.

We can now reformulate the margin constraint for sequence labeling,

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \Big) \geq 0. \quad [7.72]$$

If the score for $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ is greater than the cost-augmented score for all alternatives, then the constraint will be met. The name "cost-augmented decoding" is due to the fact that the objective includes the standard decoding problem, $\max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{w})} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}})$, plus an additional term for the cost. Essentially, we want to train against predictions that are strong and wrong: they should score highly according to the model, yet incur a large loss with respect to the ground truth. Training adjusts the weights to reduce the score of these predictions.

For cost-augmented decoding to be tractable, the cost function must decompose into local parts, just as the feature function $\boldsymbol{f}(\cdot)$ does. The Hamming cost, defined above, obeys this property. To perform cost-augmented decoding using the Hamming cost, we need only to add features $f_m(y_m) = \delta(y_m \neq y_m^{(i)})$, and assign a constant weight of 1 to these features. Decoding can then be performed using the Viterbi algorithm.[7]

As with large-margin classifiers, it is possible to formulate the learning problem in an unconstrained form, by combining a regularization term on the weights and a Lagrangian for the constraints:

$$\min_{\boldsymbol{\theta}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 - C \left(\sum_i \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \Big[\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \Big] \right), \quad [7.73]$$

In this formulation, C is a parameter that controls the tradeoff between the regularization term and the margin constraints. A number of optimization algorithms have been proposed for structured support vector machines, some of which are discussed in § 2.4.2. An empirical comparison by Kummerfeld et al. (2015) shows that stochastic subgradient descent — which is essentially a cost-augmented version of the structured perceptron — is highly competitive.

7.5.3 Conditional random fields

The **conditional random field** (CRF; Lafferty et al., 2001) is a conditional probabilistic model for sequence labeling; just as structured perceptron is built on the perceptron classifier, conditional random fields are built on the logistic regression classifier.[8] The basic

[7] Are there cost functions that do not decompose into local parts? Suppose we want to assign a constant loss c to any prediction $\hat{\boldsymbol{y}}$ in which k or more predicted tags are incorrect, and zero loss otherwise. This loss function is combinatorial over the predictions, and thus we cannot decompose it into parts.

[8] The name "conditional random field" is derived from **Markov random fields**, a general class of models in which the probability of a configuration of variables is proportional to a product of scores across pairs (or

probability model is,

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\exp(\Psi(\boldsymbol{w}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp(\Psi(\boldsymbol{w}, \boldsymbol{y}'))}. \quad [7.74]$$

This is almost identical to logistic regression (§ 2.5), but because the label space is now sequences of tags, we require efficient algorithms for both **decoding** (searching for the best tag sequence given a sequence of words $\boldsymbol{w}$ and a model $\boldsymbol{\theta}$) and for **normalization** (summing over all tag sequences). These algorithms will be based on the usual locality assumption on the scoring function, $\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m)$.

Decoding in CRFs

Decoding — finding the tag sequence $\hat{\boldsymbol{y}}$ that maximizes $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$ — is a direct application of the Viterbi algorithm. The key observation is that the decoding problem does not depend on the denominator of $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$,

$$\begin{aligned}
\hat{\boldsymbol{y}} &= \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \Psi(\boldsymbol{y}, \boldsymbol{w}) - \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp \Psi(\boldsymbol{y}', \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \Psi(\boldsymbol{y}, \boldsymbol{w}) = \operatorname*{argmax}_{\boldsymbol{y}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}).
\end{aligned}$$

This is identical to the decoding problem for structured perceptron, so the same Viterbi recurrence as defined in Equation 7.22 can be used.

Learning in CRFs

As with logistic regression, the weights $\boldsymbol{\theta}$ are learned by minimizing the regularized negative log-probability,

$$\ell = \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) \quad [7.75]$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}')\right), \quad [7.76]$$

more generally, cliques) of variables in a **factor graph**. In sequence labeling, the pairs of variables include all adjacent tags (y_m, y_{m-1}). The probability is *conditioned* on the words $\boldsymbol{w}$, which are always observed, motivating the term "conditional" in the name.

where λ controls the amount of regularization. The final term in Equation 7.76 is a sum over all possible labelings. This term is the log of the denominator in Equation 7.74, sometimes known as the **partition function**.[9] There are $|\mathcal{Y}|^M$ possible labelings of an input of size M, so we must again exploit the decomposition of the scoring function to compute this sum efficiently.

The sum $\sum_{\boldsymbol{y} \in \mathcal{Y}^{\boldsymbol{w}^{(i)}}} \exp \Psi(\boldsymbol{y}, \boldsymbol{w})$ can be computed efficiently using the **forward recurrence**, which is closely related to the Viterbi recurrence. We first define a set of **forward variables**, $\alpha_m(y_m)$, which is equal to the sum of the scores of all paths leading to tag y_m at position m:

$$\alpha_m(y_m) \triangleq \sum_{\boldsymbol{y}_{1:m-1}} \exp \sum_{n=1}^{m} s_n(y_n, y_{n-1}) \tag{7.77}$$

$$= \sum_{\boldsymbol{y}_{1:m-1}} \prod_{n=1}^{m} \exp s_n(y_n, y_{n-1}). \tag{7.78}$$

Note the similarity to the definition of the Viterbi variable, $v_m(y_m) = \max_{\boldsymbol{y}_{1:m-1}} \sum_{n=1}^{m} s_n(y_n, y_{n-1})$. In the hidden Markov model, the Viterbi recurrence had an alternative interpretation as the max-product algorithm (see Equation 7.53); analogously, the forward recurrence is known as the **sum-product algorithm**, because of the form of [7.78]. The forward variable can also be computed through a recurrence:

$$\alpha_m(y_m) = \sum_{\boldsymbol{y}_{1:m-1}} \prod_{n=1}^{m} \exp s_n(y_n, y_{n-1}) \tag{7.79}$$

$$= \sum_{y_{m-1}} (\exp s_m(y_m, y_{m-1})) \sum_{\boldsymbol{y}_{1:m-2}} \prod_{n=1}^{m-1} \exp s_n(y_n, y_{n-1}) \tag{7.80}$$

$$= \sum_{y_{m-1}} (\exp s_m(y_m, y_{m-1})) \times \alpha_{m-1}(y_{m-1}). \tag{7.81}$$

Using the forward recurrence, it is possible to compute the denominator of the conditional probability,

$$\sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{\boldsymbol{y}_{1:M}} (\exp s_{M+1}(\blacklozenge, y_M)) \prod_{m=1}^{M} \exp s_m(y_m, y_{m-1}) \tag{7.82}$$

$$= \alpha_{M+1}(\blacklozenge). \tag{7.83}$$

[9] The terminology of "potentials" and "partition functions" comes from statistical mechanics (Bishop, 2006).

The conditional log-likelihood can be rewritten,

$$\ell = \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \alpha_{M+1}(\blacklozenge). \quad [7.84]$$

Probabilistic programming environments, such as TORCH (Collobert et al., 2011) and DYNET (Neubig et al., 2017), can compute the gradient of this objective using automatic differentiation. The programmer need only implement the forward algorithm as a computation graph.

As in logistic regression, the gradient of the likelihood with respect to the parameters is a difference between observed and expected feature counts:

$$\frac{d\ell}{d\theta_j} = \lambda\theta_j + \sum_{i=1}^{N} E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] - f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}), \quad [7.85]$$

where $f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ refers to the count of feature j for token sequence $\boldsymbol{w}^{(i)}$ and tag sequence $\boldsymbol{y}^{(i)}$. The expected feature counts are computed "under the hood" when automatic differentiation is applied to Equation 7.84 (Eisner, 2016).

Before the widespread use of automatic differentiation, it was common to compute the feature expectations from marginal tag probabilities $\mathrm{p}(y_m \mid \boldsymbol{w})$. These marginal probabilities are sometimes useful on their own, and can be computed using the **forward-backward algorithm**. This algorithm combines the forward recurrence with an equivalent **backward recurrence**, which traverses the input from w_M back to w_1.

*Forward-backward algorithm

Marginal probabilities over tag bigrams can be written as,[10]

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}) = \frac{\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \exp s_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \exp s_n(y'_n, y'_{n-1})}. \quad [7.86]$$

The numerator sums over all tag sequences that include the transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$. Because we are only interested in sequences that include the tag bigram, this sum can be decomposed into three parts: the *prefixes* $\boldsymbol{y}_{1:m-1}$, terminating in $Y_{m-1} = k'$; the

[10] Recall the notational convention of upper-case letters for random variables, e.g. Y_m, and lower case letters for specific values, e.g., y_m, so that $Y_m = k$ is interpreted as the event of random variable Y_m taking the value k.

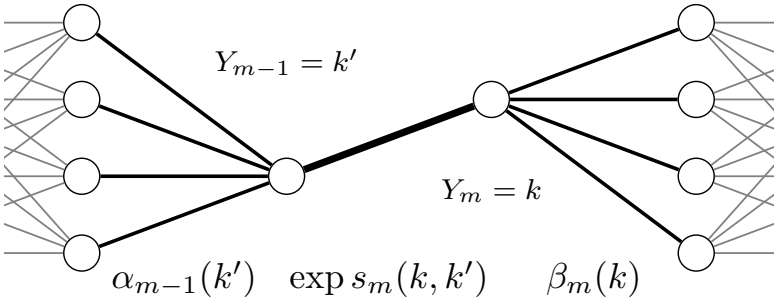


Figure 7.3: A schematic illustration of the computation of the marginal probability $\Pr(Y_{m-1} = k', Y_m = k)$, using the forward score $\alpha_{m-1}(k')$ and the backward score $\beta_m(k)$.

transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$; and the *suffixes* $\boldsymbol{y}_{m:M}$, beginning with the tag $Y_m = k$:

$$\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \exp s_n(y_n, y_{n-1}) = \sum_{\boldsymbol{y}_{1:m-1}: Y_{m-1} = k'} \prod_{n=1}^{m-1} \exp s_n(y_n, y_{n-1}) \times \exp s_m(k, k') \times \sum_{\boldsymbol{y}_{m:M}: Y_m = k} \prod_{n=m+1}^{M+1} \exp s_n(y_n, y_{n-1}). \quad [7.87]$$

The result is product of three terms: a score that sums over all the ways to get to the position $(Y_{m-1} = k')$, a score for the transition from k' to k, and a score that sums over all the ways of finishing the sequence from $(Y_m = k)$. The first term of Equation 7.87 is equal to the **forward variable**, $\alpha_{m-1}(k')$. The third term — the sum over ways to finish the sequence — can also be defined recursively, this time moving over the trellis from right to left, which is known as the **backward recurrence**:

$$\beta_m(k) \triangleq \sum_{\boldsymbol{y}_{m:M}: Y_m = k} \prod_{n=m}^{M+1} \exp s_n(y_n, y_{n-1}) \quad [7.88]$$

$$= \sum_{k' \in \mathcal{Y}} \exp s_{m+1}(k', k) \sum_{\boldsymbol{y}_{m+1:M}: Y_m = k'} \prod_{n=m+1}^{M+1} \exp s_n(y_n, y_{n-1}) \quad [7.89]$$

$$= \sum_{k' \in \mathcal{Y}} \exp s_{m+1}(k', k) \times \beta_{m+1}(k'). \quad [7.90]$$

To understand this computation, compare with the forward recurrence in Equation 7.81.

In practice, numerical stability demands that we work in the log domain,

$$\log \alpha_m(k) = \log \sum_{k' \in \mathcal{Y}} \exp\left(\log s_m(k, k') + \log \alpha_{m-1}(k')\right) \quad [7.91]$$

$$\log \beta_{m-1}(k) = \log \sum_{k' \in \mathcal{Y}} \exp\left(\log s_m(k', k) + \log \beta_m(k')\right). \quad [7.92]$$

The application of the forward and backward probabilities is shown in Figure 7.3. Both the forward and backward recurrences operate on the trellis, which implies a space complexity $\mathcal{O}(MK)$. Because both recurrences require computing a sum over K terms at each node in the trellis, their time complexity is $\mathcal{O}(MK^2)$.

7.6 Neural sequence labeling

In neural network approaches to sequence labeling, we construct a vector representation for each tagging decision, based on the word and its context. Neural networks can perform tagging as a per-token classification decision, or they can be combined with the Viterbi algorithm to tag the entire sequence globally.

7.6.1 Recurrent neural networks

Recurrent neural networks (RNNs) were introduced in chapter 6 as a language modeling technique, in which the context at token m is summarized by a recurrently-updated vector,

$$\boldsymbol{h}_m = g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}), \quad m = 1, 2, \ldots M,$$

where $\boldsymbol{x}_m$ is the vector **embedding** of the token w_m and the function g defines the recurrence. The starting condition $\boldsymbol{h}_0$ is an additional parameter of the model. The long short-term memory (LSTM) is a more complex recurrence, in which a memory cell is through a series of gates, avoiding repeated application of the non-linearity. Despite these bells and whistles, both models share the basic architecture of recurrent updates across a sequence, and both will be referred to as RNNs here.

A straightforward application of RNNs to sequence labeling is to score each tag y_m as a linear function of $\boldsymbol{h}_m$:

$$\psi_m(y) = \boldsymbol{\beta}_y \cdot \boldsymbol{h}_m \quad [7.93]$$

$$\hat{y}_m = \operatorname*{argmax}_y \psi_m(y). \quad [7.94]$$

The score $\psi_m(y)$ can also be converted into a probability distribution using the usual softmax operation,

$$\mathrm{p}(y \mid \boldsymbol{w}_{1:m}) = \frac{\exp \psi_m(y)}{\sum_{y' \in \mathcal{Y}} \exp \psi_m(y')}. \quad [7.95]$$

Using this transformation, it is possible to train the tagger from the negative log-likelihood of the tags, as in a conditional random field. Alternatively, a hinge loss or margin loss objective can be constructed from the raw scores $\psi_m(y)$.

The hidden state $\boldsymbol{h}_m$ accounts for information in the input leading up to position m, but it ignores the subsequent tokens, which may also be relevant to the tag y_m. This can be addressed by adding a second RNN, in which the input is reversed, running the recurrence from w_M to w_1. This is known as a **bidirectional recurrent neural network** (Graves and Schmidhuber, 2005), and is specified as:

$$\overleftarrow{\boldsymbol{h}}_m = g(\boldsymbol{x}_m, \overleftarrow{\boldsymbol{h}}_{m+1}), \quad m = 1, 2, \ldots, M. \tag{7.96}$$

The hidden states of the left-to-right RNN are denoted $\overrightarrow{\boldsymbol{h}}_m$. The left-to-right and right-to-left vectors are concatenated, $\boldsymbol{h}_m = [\overleftarrow{\boldsymbol{h}}_m; \overrightarrow{\boldsymbol{h}}_m]$. The scoring function in Equation 7.93 is applied to this concatenated vector.

Bidirectional RNN tagging has several attractive properties. Ideally, the representation $\boldsymbol{h}_m$ summarizes the useful information from the surrounding context, so that it is not necessary to design explicit features to capture this information. If the vector $\boldsymbol{h}_m$ is an adequate summary of this context, then it may not even be necessary to perform the tagging jointly: in general, the gains offered by joint tagging of the entire sequence are diminished as the individual tagging model becomes more powerful. Using backpropagation, the word vectors $\boldsymbol{x}$ can be trained "end-to-end", so that they capture word properties that are useful for the tagging task. Alternatively, if limited labeled data is available, we can use word embeddings that are "pre-trained" from unlabeled data, using a language modeling objective (as in § 6.3) or a related word embedding technique (see chapter 14). It is even possible to combine both fine-tuned and pre-trained embeddings in a single model.

Neural structure prediction The bidirectional recurrent neural network incorporates information from throughout the input, but each tagging decision is made independently. In some sequence labeling applications, there are very strong dependencies between tags: it may even be impossible for one tag to follow another. In such scenarios, the tagging decision must be made jointly across the entire sequence.

Neural sequence labeling can be combined with the Viterbi algorithm by defining the local scores as:

$$s_m(y_m, y_{m-1}) = \boldsymbol{\beta}_{y_m} \cdot \boldsymbol{h}_m + \eta_{y_{m-1}, y_m}, \tag{7.97}$$

where $\boldsymbol{h}_m$ is the RNN hidden state, $\boldsymbol{\beta}_{y_m}$ is a vector associated with tag y_m, and η_{y_{m-1}, y_m} is a scalar parameter for the tag transition (y_{m-1}, y_m). These local scores can then be incorporated into the Viterbi algorithm for inference, and into the forward algorithm for training. This model is shown in Figure 7.4. It can be trained from the conditional log-likelihood objective defined in Equation 7.76, backpropagating to the tagging parameters

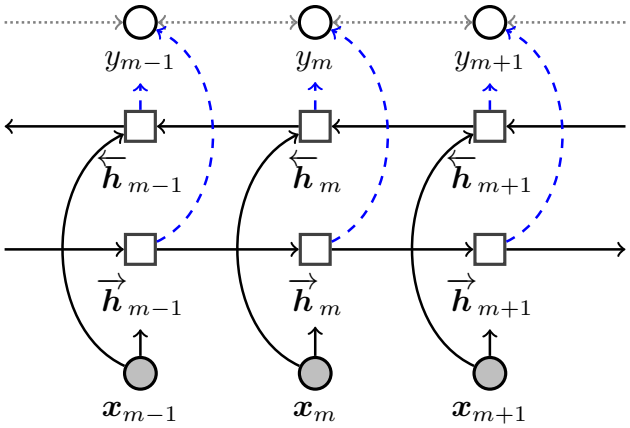


Figure 7.4: **Bidirectional LSTM** for sequence labeling. The solid lines indicate computation, the dashed lines indicate probabilistic dependency, and the dotted lines indicate the optional additional probabilistic dependencies between labels in the biLSTM-CRF.

$\boldsymbol{\beta}$ and $\boldsymbol{\eta}$, as well as the parameters of the RNN. This model is called the **LSTM-CRF**, due to its combination of aspects of the long short-term memory and conditional random field models (Huang et al., 2015).

The LSTM-CRF is especially effective on the task of **named entity recognition** (Lample et al., 2016), a sequence labeling task that is described in detail in § 8.3. This task has strong dependencies between adjacent tags, so structure prediction is especially important.

7.6.2 Character-level models

As in language modeling, rare and unseen words are a challenge: if we encounter a word that was not in the training data, then there is no obvious choice for the word embedding $\boldsymbol{x}_m$. One solution is to use a generic **unseen word** embedding for all such words. However, in many cases, properties of unseen words can be guessed from their spellings. For example, *whimsical* does not appear in the Universal Dependencies (UD) English Treebank, yet the suffix *-al* makes it likely to be adjective; by the same logic, *unflinchingly* is likely to be an adverb, and *barnacle* is likely to be a noun.

In feature-based models, these morphological properties were handled by suffix features; in a neural network, they can be incorporated by constructing the embeddings of unseen words from their spellings or morphology. One way to do this is to incorporate an additional layer of bidirectional RNNs, one for each word in the vocabulary (Ling et al., 2015). For each such character-RNN, the inputs are the characters, and the output is the concatenation of the final states of the left-facing and right-facing passes, $\boldsymbol{\phi}_w =$

$[\overrightarrow{\boldsymbol{h}}^{(w)}_{N_w}; \overleftarrow{\boldsymbol{h}}^{(w)}_{0}]$, where $\overrightarrow{\boldsymbol{h}}^{(w)}_{N_w}$ is the final state of the right-facing pass for word w, and N_w is the number of characters in the word. The character RNN model is trained by backpropagation from the tagging objective. On the test data, the trained RNN is applied to out-of-vocabulary words (or all words), yielding inputs to the word-level tagging RNN. Other approaches to compositional word embeddings are described in § 14.7.1.

7.6.3 Convolutional Neural Networks for Sequence Labeling

One disadvantage of recurrent neural networks is that the architecture requires iterating through the sequence of inputs and predictions: each hidden vector $\boldsymbol{h}_m$ must be computed from the previous hidden vector $\boldsymbol{h}_{m-1}$, before predicting the tag y_m. These iterative computations are difficult to parallelize, and fail to exploit the speedups offered by **graphics processing units (GPUs)** on operations such as matrix multiplication. **Convolutional neural networks** achieve better computational performance by predicting each label y_m from a set of matrix operations on the neighboring word embeddings, $\boldsymbol{x}_{m-k:m+k}$ (Collobert et al., 2011). Because there is no hidden state to update, the predictions for each y_m can be computed in parallel. For more on convolutional neural networks, see § 3.4. Character-based word embeddings can also be computed using convolutional neural networks (Santos and Zadrozny, 2014).

7.7 *Unsupervised sequence labeling

In unsupervised sequence labeling, the goal is to induce a hidden Markov model from a corpus of *unannotated* text $(\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \ldots, \boldsymbol{w}^{(N)})$, where each $\boldsymbol{w}^{(i)}$ is a sequence of length $M^{(i)}$. This is an example of the general problem of **structure induction**, which is the unsupervised version of structure prediction. The tags that result from unsupervised sequence labeling might be useful for some downstream task, or they might help us to better understand the language's inherent structure. For part-of-speech tagging, it is common to use a tag dictionary that lists the allowed tags for each word, simplifying the problem (Christodoulopoulos et al., 2010).

Unsupervised learning in hidden Markov models can be performed using the **Baum-Welch algorithm**, which combines the forward-backward algorithm (§ 7.5.3) with expectation-maximization (EM; § 5.1.2). In the M-step, the HMM parameters from expected counts:

$$\Pr(W = i \mid Y = k) = \phi_{k,i} = \frac{E[\text{count}(W = i, Y = k)]}{E[\text{count}(Y = k)]}$$
$$\Pr(Y_m = k \mid Y_{m-1} = k') = \lambda_{k',k} = \frac{E[\text{count}(Y_m = k, Y_{m-1} = k')]}{E[\text{count}(Y_{m-1} = k')]}$$

The expected counts are computed in the E-step, using the forward and backward recurrences. The local scores follow the usual definition for hidden Markov models,

$$s_m(k, k') = \log \mathrm{p}_E(w_m \mid Y_m = k; \boldsymbol{\phi}) + \log \mathrm{p}_T(Y_m = k \mid Y_{m-1} = k'; \lambda). \quad [7.98]$$

The expected transition counts for a single instance are,

$$E[\text{count}(Y_m = k, Y_{m-1} = k') \mid \boldsymbol{w}] = \sum_{m=1}^{M} \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}) \quad [7.99]$$

$$= \frac{\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \exp s_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \exp s_n(y'_n, y'_{n-1})}. \quad [7.100]$$

As described in § 7.5.3, these marginal probabilities can be computed from the forward-backward recurrence,

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}) = \frac{\alpha_{m-1}(k') \times \exp s_m(k, k') \times \beta_m(k)}{\alpha_{M+1}(\blacklozenge)}. \quad [7.101]$$

In a hidden Markov model, each element of the forward-backward computation has a special interpretation:

$$\alpha_{m-1}(k') = \mathrm{p}(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) \quad [7.102]$$

$$\exp s_m(k, k') = \mathrm{p}(Y_m = k, w_m \mid Y_{m-1} = k') \quad [7.103]$$

$$\beta_m(k) = \mathrm{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k). \quad [7.104]$$

Applying the conditional independence assumptions of the hidden Markov model (defined in Algorithm 12), the product is equal to the joint probability of the tag bigram and the entire input,

$$\begin{aligned}\alpha_{m-1}(k') \times \exp s_m(k, k') \times \beta_m(k) =& \mathrm{p}(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) \\ & \times \mathrm{p}(Y_m = k, w_m \mid Y_{m-1} = k') \\ & \times \mathrm{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k) \\ =& \mathrm{p}(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M}). \end{aligned} \quad [7.105]$$

Dividing by $\alpha_{M+1}(\blacklozenge) = \mathrm{p}(\boldsymbol{w}_{1:M})$ gives the desired probability,

$$\frac{\alpha_{m-1}(k') \times s_m(k, k') \times \beta_m(k)}{\alpha_{M+1}(\blacklozenge)} = \frac{\mathrm{p}(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M})}{\mathrm{p}(\boldsymbol{w}_{1:M})} \quad [7.106]$$

$$= \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M}). \quad [7.107]$$

The expected emission counts can be computed in a similar manner, using the product $\alpha_m(k) \times \beta_m(k)$.

7.7.1 Linear dynamical systems

The forward-backward algorithm can be viewed as Bayesian state estimation in a discrete state space. In a continuous state space, $\boldsymbol{y}_m \in \mathbb{R}^K$, the equivalent algorithm is the **Kalman smoother**. It also computes marginals $\text{p}(\boldsymbol{y}_m \mid \boldsymbol{x}_{1:M})$, using a similar two-step algorithm of forward and backward passes. Instead of computing a trellis of values at each step, the Kalman smoother computes a probability density function $q_{\boldsymbol{y}_m}(\boldsymbol{y}_m; \boldsymbol{\mu}_m, \Sigma_m)$, characterized by a mean $\boldsymbol{\mu}_m$ and a covariance Σ_m around the latent state. Connections between the Kalman smoother and the forward-backward algorithm are elucidated by Minka (1999) and Murphy (2012).

7.7.2 Alternative unsupervised learning methods

As noted in § 5.5, expectation-maximization is just one of many techniques for structure induction. One alternative is to use **Markov Chain Monte Carlo (MCMC)** sampling algorithms, which are briefly described in § 5.5.1. For the specific case of sequence labeling, Gibbs sampling can be applied by iteratively sampling each tag y_m conditioned on all the others (Finkel et al., 2005):

$$\text{p}(y_m \mid \boldsymbol{y}_{-m}, \boldsymbol{w}_{1:M}) \propto \text{p}(w_m \mid y_m)\text{p}(y_m \mid \boldsymbol{y}_{-m}). \quad [7.108]$$

Gibbs Sampling has been applied to unsupervised part-of-speech tagging by Goldwater and Griffiths (2007). **Beam sampling** is a more sophisticated sampling algorithm, which randomly draws entire sequences $\boldsymbol{y}_{1:M}$, rather than individual tags y_m; this algorithm was applied to unsupervised part-of-speech tagging by Van Gael et al. (2009). Spectral learning (see § 5.5.2) can also be applied to sequence labeling. By factoring matrices of co-occurrence counts of word bigrams and trigrams (Song et al., 2010; Hsu et al., 2012), it is possible to obtain globally optimal estimates of the transition and emission parameters, under mild assumptions.

7.7.3 Semiring notation and the generalized viterbi algorithm

The Viterbi and Forward recurrences can each be performed over probabilities or log probabilities, yielding a total of four closely related recurrences. These four recurrence scan in fact be expressed as a single recurrence in a more general notation, known as **semiring algebra**. Let the symbols $\oplus$ and $\otimes$ represent generalized addition and multiplication respectively.[11] Given these operators, a generalized Viterbi recurrence is denoted,

$$v_m(k) = \bigoplus_{k' \in \mathcal{Y}} s_m(k, k') \otimes v_{m-1}(k'). \quad [7.109]$$

[11] In a semiring, the addition and multiplication operators must both obey associativity, and multiplication must distribute across addition; the addition operator must be commutative; there must be additive and multiplicative identities $\overline{0}$ and $\overline{1}$, such that $a \oplus \overline{0} = a$ and $a \otimes \overline{1} = a$; and there must be a multiplicative annihilator $\overline{0}$, such that $a \otimes \overline{0} = \overline{0}$.

Each recurrence that we have seen so far is a special case of this generalized Viterbi recurrence:

- In the max-product Viterbi recurrence over probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to multiplication.
- In the forward recurrence over probabilities, the $\oplus$ operation corresponds to addition, and the $\otimes$ operation corresponds to multiplication.
- In the max-product Viterbi recurrence over log-probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to addition.[12]
- In the forward recurrence over log-probabilities, the $\oplus$ operation corresponds to log-addition, $a \oplus b = \log(e^a + e^b)$. The $\otimes$ operation corresponds to addition.

The mathematical abstraction offered by semiring notation can be applied to the software implementations of these algorithms, yielding concise and modular implementations. For example, in the OpenFST library, generic operations are parametrized by the choice of semiring (Allauzen et al., 2007).

Exercises

1. Extend the example in § 7.3.1 to the sentence *they can can fish*, meaning that "they can put fish into cans." Build the trellis for this example using the weights in Table 7.1, and identify the best-scoring tag sequence. If the scores for noun and verb are tied, then you may assume that the backpointer always goes to noun.

2. Using the tagset $\mathcal{Y} = \{N, V\}$, and the feature set $\boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \{(w_m, y_m), (y_m, y_{m-1})\}$, show that there is no set of weights that give the correct tagging for both *they can fish* (N V V) and *they can can fish* (N V V N).

3. Work out what happens if you train a structured perceptron on the two examples mentioned in the previous problem, using the transition and emission features (y_m, y_{m-1}) and (y_m, w_m). Initialize all weights at 0, and assume that the Viterbi algorithm always chooses N when the scores for the two tags are tied, so that the initial prediction for *they can fish* is N N N.

4. Consider the garden path sentence, *The old man the boat.* Given word-tag and tag-tag features, what inequality in the weights must hold for the correct tag sequence to outscore the garden path tag sequence for this example?

[12] This is sometimes called the **tropical semiring**, in honor of the Brazilian mathematician Imre Simon.

5. Using the weights in Table 7.1, explicitly compute the log-probabilities for all possible taggings of the input *fish can*. Verify that the forward algorithm recovers the aggregate log probability.

6. Sketch out an algorithm for a variant of Viterbi that returns the top-n label sequences. What is the time and space complexity of this algorithm?

7. Show how to compute the marginal probability $\Pr(y_{m-2} = k, y_m = k' \mid \boldsymbol{w}_{1:M})$, in terms of the forward and backward variables, and the potentials $s_n(y_n, y_{n-1})$.

8. Suppose you receive a stream of text, where some of tokens have been replaced at random with *NOISE*. For example:

   - Source: *I try all things, I achieve what I can*
   - Message received: *I try NOISE NOISE, I NOISE what I NOISE*

   Assume you have access to a pre-trained bigram language model, which gives probabilities $\text{p}(w_m \mid w_{m-1})$. These probabilities can be assumed to be non-zero for all bigrams.

   Show how to use the Viterbi algorithm to recover the source by maximizing the bigram language model log-probability. Specifically, set the scores $s_m(y_m, y_{m-1})$ so that the Viterbi algorithm selects a sequence of words that maximizes the bigram language model log-probability, while leaving the non-noise tokens intact. Your solution should not modify the logic of the Viterbi algorithm, it should only set the scores $s_m(y_m, y_{m-1})$.

9. Let $\alpha(\cdot)$ and $\beta(\cdot)$ indicate the forward and backward variables as defined in § 7.5.3. Prove that $\alpha_{M+1}(\blacklozenge) = \beta_0(\lozenge) = \sum_y \alpha_m(y)\beta_m(y), \forall m \in \{1, 2, \ldots, M\}$.

10. Consider an RNN tagging model with a $\tanh$ activation function on the hidden layer, and a hinge loss on the output. (The problem also works for the margin loss and negative log-likelihood.) Suppose you initialize all parameters to zero: this includes the word embeddings that make up $\boldsymbol{x}$, the transition matrix Θ, the output weights $\boldsymbol{\beta}$, and the initial hidden state $\boldsymbol{h}_0$.

    a) Prove that for any data and for any gradient-based learning algorithm, all parameters will be stuck at zero.

    b) Would a sigmoid activation function avoid this problem?

Chapter 8

Applications of sequence labeling

Sequence labeling has applications throughout natural language processing. This chapter focuses on part-of-speech tagging, morpho-syntactic attribute tagging, named entity recognition, and tokenization. It also touches briefly on two applications to interactive settings: dialogue act recognition and the detection of code-switching points between languages.

8.1 Part-of-speech tagging

The **syntax** of a language is the set of principles under which sequences of words are judged to be grammatically acceptable by fluent speakers. One of the most basic syntactic concepts is the **part-of-speech** (POS), which refers to the syntactic role of each word in a sentence. This concept was used informally in the previous chapter, and you may have some intuitions from your own study of English. For example, in the sentence *We like vegetarian sandwiches*, you may already know that *we* and *sandwiches* are nouns, *like* is a verb, and *vegetarian* is an adjective. These labels depend on the context in which the word appears: in *she eats like a vegetarian*, the word *like* is a preposition, and the word *vegetarian* is a noun.

Parts-of-speech can help to disentangle or explain various linguistic problems. Recall Chomsky's proposed distinction in chapter 6:

(8.1) a. Colorless green ideas sleep furiously.
b. * Ideas colorless furiously green sleep.

One difference between these two examples is that the first contains part-of-speech transitions that are typical in English: adjective to adjective, adjective to noun, noun to verb, and verb to adverb. The second example contains transitions that are unusual: noun to adjective and adjective to verb. The ambiguity in a headline like,

(8.2) Teacher Strikes Idle Children

can also be explained in terms of parts of speech: in the interpretation that was likely intended, *strikes* is a noun and *idle* is a verb; in the alternative explanation, *strikes* is a verb and *idle* is an adjective.

Part-of-speech tagging is often taken as a early step in a natural language processing pipeline. Indeed, parts-of-speech provide features that can be useful for many of the tasks that we will encounter later, such as parsing (chapter 10), coreference resolution (chapter 15), and relation extraction (chapter 17).

8.1.1 Parts-of-Speech

The **Universal Dependencies** project (UD) is an effort to create syntactically-annotated corpora across many languages, using a single annotation standard (Nivre et al., 2016). As part of this effort, they have designed a part-of-speech **tagset**, which is meant to capture word classes across as many languages as possible.[1] This section describes that inventory, giving rough definitions for each of tags, along with supporting examples.

Part-of-speech tags are **morphosyntactic**, rather than semantic, categories. This means that they describe words in terms of how they pattern together and how they are internally constructed (e.g., what suffixes and prefixes they include). For example, you may think of a noun as referring to objects or concepts, and verbs as referring to actions or events. But events can also be nouns:

(8.3) . . . the **howling** of the **shrieking** storm.

Here *howling* and *shrieking* are events, but grammatically they act as a noun and adjective respectively.

The Universal Dependency part-of-speech tagset

The UD tagset is broken up into three groups: open class tags, closed class tags, and "others."

Open class tags Nearly all languages contain nouns, verbs, adjectives, and adverbs.[2] These are all **open word classes**, because new words can easily be added to them. The UD tagset includes two other tags that are open classes: proper nouns and interjections.

- **Nouns** (UD tag: NOUN) tend to describe entities and concepts, e.g.,

[1] The UD tagset builds on earlier work from Petrov et al. (2012), in which a set of twelve universal tags was identified by creating mappings from tagsets for individual languages.

[2] One prominent exception is Korean, which some linguists argue does not have adjectives Kim (2002).

(8.4) **Toes** are scarce among veteran **blubber men**.

In English, nouns tend to follow determiners and adjectives, and can play the subject role in the sentence. They can be marked for the plural number by an *-s* suffix.

- **Proper nouns** (PROPN) are tokens in names, which uniquely specify a given entity,

  (8.5) "**Moby Dick**?" shouted **Ahab**.

- **Verbs** (VERB), according to the UD guidelines, "typically signal events and actions." But they are also defined grammatically: they "can constitute a minimal predicate in a clause, and govern the number and types of other constituents which may occur in a clause."[3]

  (8.6) "Moby Dick?" **shouted** Ahab.

  (8.7) Shall we **keep chasing** this murderous fish?

  English verbs tend to come in between the subject and some number of direct objects, depending on the verb. They can be marked for **tense** and **aspect** using suffixes such as *-ed* and *-ing*. (These suffixes are an example of **inflectional morphology**, which is discussed in more detail in § 9.1.4.)

- **Adjectives** (ADJ) describe properties of entities,

  (8.8) a. Shall we keep chasing this **murderous** fish?
  b. Toes are **scarce** among **veteran** blubber men.

  In the second example, *scarce* is a predicative adjective, linked to the subject by the **copula verb** *are*. In contrast, *murderous* and *veteran* are attributive adjectives, modifying the noun phrase in which they are embedded.

- **Adverbs** (ADV) describe properties of events, and may also modify adjectives or other adverbs:

  (8.9) a. It is not down on any map; true places **never** are.
  b. ... **treacherously** hidden beneath the loveliest tints of azure
  c. Not drowned **entirely**, though.

- **Interjections** (INTJ) are used in exclamations, e.g.,

  (8.10) **Aye aye**! it was that accursed white whale that razed me.

[3] http://universaldependencies.org/u/pos/VERB.html

Closed class tags Closed word classes rarely receive new members. They are sometimes referred to as **function words** — as opposed to **content words** — as they have little lexical meaning of their own, but rather, help to organize the components of the sentence.

- **Adpositions** (ADP) describe the relationship between a complement (usually a noun phrase) and another unit in the sentence, typically a noun or verb phrase.

  (8.11) a. Toes are scarce **among** veteran blubber men.
  b. It is not **down on** any map.
  c. Give not thyself **up** then.

  As the examples show, English generally uses prepositions, which are adpositions that appear before their complement. (An exception is *ago*, as in, *we met three days **ago***). Postpositions are used in other languages, such as Japanese and Turkish.
- **Auxiliary verbs** (AUX) are a closed class of verbs that add information such as tense, aspect, person, and number.

  (8.12) a. **Shall** we keep chasing this murderous fish?
  b. What the white whale was to Ahab, **has been** hinted.
  c. Ahab **must** use tools.
  d. Meditation and water **are** wedded forever.
  e. Toes **are** scarce among veteran blubber men.

  The final example is a copula verb, which is also tagged as an auxiliary in the UD corpus.
- **Coordinating conjunctions** (CCONJ) express relationships between two words or phrases, which play a parallel role:

  (8.13) Meditation **and** water are wedded forever.

- **Subordinating conjunctions** (SCONJ) link two clauses, making one syntactically subordinate to the other:

  (8.14) It is the easiest thing in the world for a man to look as **if** he had a great secret in him.

  Note that
- **Pronouns** (PRON) are words that substitute for nouns or noun phrases.

  (8.15) a. Be **it what it** will, **I**'ll go to **it** laughing.

b. **I** try all things, **I** achieve **what I** can.

The example includes the personal pronouns *I* and *it*, as well as the relative pronoun *what*. Other pronouns include *myself*, *somebody*, and *nothing*.

- **Determiners** (DET) provide additional information about the nouns or noun phrases that they modify:

(8.16) a. What **the** white whale was to Ahab, has been hinted.
b. It is not down on **any** map.
c. I try **all** things ...
d. Shall we keep chasing **this** murderous fish?

Determiners include articles (*the*), possessive determiners (*their*), demonstratives (*this murderous fish*), and quantifiers (*any map*).

- **Numerals** (NUM) are an infinite but closed class, which includes integers, fractions, and decimals, regardless of whether spelled out or written in numerical form.

(8.17) a. How then can this **one** small heart beat.
b. I am going to put him down for the **three hundredth**.

- **Particles** (PART) are a catch-all of function words that combine with other words or phrases, but do not meet the conditions of the other tags. In English, this includes the infinitival *to*, the possessive marker, and negation.

(8.18) a. Better **to** sleep with a sober cannibal than a drunk Christian.
b. So man**'s** insanity is heaven**'s** sense
c. It is **not** down on any map

As the second example shows, the possessive marker is not considered part of the same token as the word that it modifies, so that *man's* is split into two tokens. (Tokenization is described in more detail in § 8.4.) A non-English example of a particle is the Japanese question marker *ka*:[4]

(8.19) *Sensei desu ka*
Teacher is ?
Is she a teacher?

[4]In this notation, the first line is the transliterated Japanese text, the second line is a token-to-token **gloss**, and the third line is the translation.

Other The remaining UD tags include punctuation (PUN) and symbols (SYM). Punctuation is purely structural — e.g., commas, periods, colons — while symbols can carry content of their own. Examples of symbols include dollar and percentage symbols, mathematical operators, emoticons, emojis, and internet addresses. A final catch-all tag is X, which is used for words that cannot be assigned another part-of-speech category. The X tag is also used in cases of **code switching** (between languages), described in § 8.5.

Other tagsets

Prior to the Universal Dependency treebank, part-of-speech tagging was performed using language-specific tagsets. The dominant tagset for English was designed as part of the **Penn Treebank** (PTB), and it includes 45 tags — more than three times as many as the UD tagset. This granularity is reflected in distinctions between singular and plural nouns, verb tenses and aspects, possessive and non-possessive pronouns, comparative and superlative adjectives and adverbs (e.g., *faster*, *fastest*), and so on. The Brown corpus includes a tagset that is even more detailed, with 87 tags (Francis, 1964), including special tags for individual auxiliary verbs such as *be*, *do*, and *have*.

Different languages make different distinctions, and so the PTB and Brown tagsets are not appropriate for a language such as Chinese, which does not mark the verb tense (Xia, 2000); nor for Spanish, which marks every combination of person and number in the verb ending; nor for German, which marks the case of each noun phrase. Each of these languages requires more detail than English in some areas of the tagset, and less in other areas. The strategy of the Universal Dependencies corpus is to design a coarse-grained tagset to be used across all languages, and then to additionally annotate language-specific **morphosyntactic attributes**, such as number, tense, and case. The attribute tagging task is described in more detail in § 8.2.

Social media such as Twitter have been shown to require tagsets of their own (Gimpel et al., 2011). Such corpora contain some tokens that are not equivalent to anything encountered in a typical written corpus: e.g., emoticons, URLs, and hashtags. Social media also includes dialectal words like *gonna* ('going to', e.g. *We gonna be fine*) and *Ima* ('I'm going to', e.g., *Ima tell you one more time*), which can be analyzed either as non-standard orthography (making tokenization impossible), or as lexical items in their own right. In either case, it is clear that existing tags like NOUN and VERB cannot handle cases like *Ima*, which combine aspects of the noun and verb. Gimpel et al. (2011) therefore propose a new set of tags to deal with these cases.

8.1.2 Accurate part-of-speech tagging

Part-of-speech tagging is the problem of selecting the correct tag for each word in a sentence. Success is typically measured by accuracy on an annotated test set, which is simply the fraction of tokens that were tagged correctly.

Baselines

A simple baseline for part-of-speech tagging is to choose the most common tag for each word. For example, in the Universal Dependencies treebank, the word *talk* appears 96 times, and 85 of those times it is labeled as a VERB: therefore, this baseline will always predict VERB for this word. For words that do not appear in the training corpus, the baseline simply guesses the most common tag overall, which is NOUN. In the Penn Treebank, this simple baseline obtains accuracy above 92%. A more rigorous evaluation is the accuracy on **out-of-vocabulary words**, which are not seen in the training data. Tagging these words correctly requires attention to the context and the word's internal structure.

Contemporary approaches

Conditional random fields and structured perceptron perform at or near the state-of-the-art for part-of-speech tagging in English. For example, (Collins, 2002) achieved 97.1% accuracy on the Penn Treebank, using a structured perceptron with the following base features (originally introduced by Ratnaparkhi (1996)):

- current word, w_m
- previous words, w_{m-1}, w_{m-2}
- next words, w_{m+1}, w_{m+2}
- previous tag, y_{m-1}
- previous two tags, (y_{m-1}, y_{m-2})
- for rare words:
  - first k characters, up to $k = 4$
  - last k characters, up to $k = 4$
  - whether w_m contains a number, uppercase character, or hyphen.

Similar results for the PTB data have been achieved using conditional random fields (CRFs; Toutanova et al., 2003).

More recent work has demonstrated the power of neural sequence models, such as the **long short-term memory (LSTM)** (§ 7.6). Plank et al. (2016) apply a CRF and a bidirectional LSTM to twenty-two languages in the UD corpus, achieving an average accuracy of 94.3% for the CRF, and 96.5% with the bi-LSTM. Their neural model employs three types of embeddings: fine-tuned word embeddings, which are updated during training; pre-trained word embeddings, which are never updated, but which help to tag out-of-vocabulary words; and character-based embeddings. The character-based embeddings are computed by running an LSTM on the individual characters in each word, thereby capturing common orthographic patterns such as prefixes, suffixes, and capitalization. Extensive evaluations show that these additional embeddings are crucial to their model's success.

word	PTB tag	UD tag	UD attributes
The	DT	DET	DEFINITE=DEF PRONTYPE=ART
German	JJ	ADJ	DEGREE=POS
Expressionist	NN	NOUN	NUMBER=SING
movement	NN	NOUN	NUMBER=SING
was	VBD	AUX	MOOD=IND NUMBER=SING PERSON=3 TENSE=PAST VERBFORM=FIN
destroyed	VBN	VERB	TENSE=PAST VERBFORM=PART VOICE=PASS
as	IN	ADP	
a	DT	DET	DEFINITE=IND PRONTYPE=ART
result	NN	NOUN	NUMBER=SING
.	.	PUNCT	

Figure 8.1: UD and PTB part-of-speech tags, and UD morphosyntactic attributes. Example selected from the UD 1.4 English corpus.

8.2 Morphosyntactic Attributes

There is considerably more to say about a word than whether it is a noun or a verb: in English, verbs are distinguish by features such tense and aspect, nouns by number, adjectives by degree, and so on. These features are language-specific: other languages distinguish other features, such as **case** (the role of the noun with respect to the action of the sentence, which is marked in languages such as Latin and German[5]) and **evidentiality** (the source of information for the speaker's statement, which is marked in languages such as Turkish). In the UD corpora, these attributes are annotated as feature-value pairs for each token.[6]

An example is shown in Figure 8.1. The determiner *the* is marked with two attributes: PRONTYPE=ART, which indicates that it is an **article** (as opposed to another type of deter-

[5]Case is marked in English for some personal pronouns, e.g., ***She*** *saw* ***her***, *They* saw *them*.

[6]The annotation and tagging of morphosyntactic attributes can be traced back to earlier work on Turkish (Oflazer and Kuruöz, 1994) and Czech (Hajič and Hladká, 1998). MULTEXT-East was an early multilingual corpus to include morphosyntactic attributes (Dimitrova et al., 1998).

miner or pronominal modifier), and DEFINITE=DEF, which indicates that it is a **definite article** (referring to a specific, known entity). The verbs are each marked with several attributes. The auxiliary verb *was* is third-person, singular, past tense, finite (conjugated), and indicative (describing an event that has happened or is currently happenings); the main verb *destroyed* is in participle form (so there is no additional person and number information), past tense, and passive voice. Some, but not all, of these distinctions are reflected in the PTB tags VBD (past-tense verb) and VBN (past participle).

While there are thousands of papers on part-of-speech tagging, there is comparatively little work on automatically labeling morphosyntactic attributes. Faruqui et al. (2016) train a support vector machine classification model, using a minimal feature set that includes the word itself, its prefixes and suffixes, and type-level information listing all possible morphosyntactic attributes for each word and its neighbors. Mueller et al. (2013) use a conditional random field (CRF), in which the tag space consists of all observed combinations of morphosyntactic attributes (e.g., the tag would be DEF+ART for the word *the* in Figure 8.1). This massive tag space is managed by decomposing the feature space over individual attributes, and pruning paths through the trellis. More recent work has employed bidirectional LSTM sequence models. For example, Pinter et al. (2017) train a bidirectional LSTM sequence model. The input layer and hidden vectors in the LSTM are shared across attributes, but each attribute has its own output layer, culminating in a softmax over all attribute values, e.g. $y_t^{\text{NUMBER}} \in \{\text{SING}, \text{PLURAL}, \ldots\}$. They find that character-level information is crucial, especially when the amount of labeled data is limited.

Evaluation is performed by first computing recall and precision for each attribute. These scores can then be averaged at either the type or token level to obtain micro- or macro-F-MEASURE. Pinter et al. (2017) evaluate on 23 languages in the UD treebank, reporting a median micro-F-MEASURE of 0.95. Performance is strongly correlated with the size of the labeled dataset for each language, with a few outliers: for example, Chinese is particularly difficult, because although the dataset is relatively large (10^5 tokens in the UD 1.4 corpus), only 6% of tokens have any attributes, offering few useful labeled instances.

8.3 Named Entity Recognition

A classical problem in information extraction is to recognize and extract mentions of **named entities** in text. In news documents, the core entity types are people, locations, and organizations; more recently, the task has been extended to include amounts of money, percentages, dates, and times. In item 8.20a (Figure 8.2), the named entities include: *The U.S. Army*, an organization; *Atlanta*, a location; and *May 14, 1864*, a date. Named entity recognition is also a key task in **biomedical natural language processing**, with entity types including proteins, DNA, RNA, and cell lines (e.g., Collier et al., 2000; Ohta et al., 2002). Figure 8.2 shows an example from the GENIA corpus of biomedical research ab-

(8.20) a. *The U.S. Army captured Atlanta on May 14 , 1864*
B-ORG I-ORG I-ORG O B-LOC O B-DATE I-DATE I-DATE I-DATE

b. *Number of glucocorticoid receptors in lymphocytes and ...*
O O B-PROTEIN I-PROTEIN O B-CELLTYPE O ...

Figure 8.2: BIO notation for named entity recognition. Example (8.20b) is drawn from the GENIA corpus of biomedical documents (Ohta et al., 2002).

stracts.

A standard approach to tagging named entity spans is to use discriminative sequence labeling methods such as conditional random fields. However, the named entity recognition (NER) task would seem to be fundamentally different from sequence labeling tasks like part-of-speech tagging: rather than tagging each token, the goal in is to recover *spans* of tokens, such as *The United States Army*.

This is accomplished by the **BIO notation**, shown in Figure 8.2. Each token at the beginning of a name span is labeled with a B- prefix; each token within a name span is labeled with an I- prefix. These prefixes are followed by a tag for the entity type, e.g. B-LOC for the beginning of a location, and I-PROTEIN for the inside of a protein name. Tokens that are not parts of name spans are labeled as O. From this representation, the entity name spans can be recovered unambiguously. This tagging scheme is also advantageous for learning: tokens at the beginning of name spans may have different properties than tokens within the name, and the learner can exploit this. This insight can be taken even further, with special labels for the **l**ast tokens of a name span, and for **u**nique tokens in name spans, such as *Atlanta* in the example in Figure 8.2. This is called BILOU notation, and it can yield improvements in supervised named entity recognition (Ratinov and Roth, 2009).

Feature-based sequence labeling Named entity recognition was one of the first applications of conditional random fields (McCallum and Li, 2003). The use of Viterbi decoding restricts the feature function $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$ to be a sum of local features, $\sum_m \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$, so that each feature can consider only local adjacent tags. Typical features include tag transitions, word features for w_m and its neighbors, character-level features for prefixes and suffixes, and "word shape" features for capitalization and other orthographic properties. As an example, base features for the word *Army* in the example in (8.20a) include:

$$(\text{CURR-WORD:}\textit{Army}, \text{PREV-WORD:}\textit{U.S.}, \text{NEXT-WORD:}\textit{captured}, \text{PREFIX-1:}\textit{A-},$$
$$\text{PREFIX-2:}\textit{Ar-}, \text{SUFFIX-1:}\textit{-y}, \text{SUFFIX-2:}\textit{-my}, \text{SHAPE:}\textit{Xxxx})$$

Features can also be obtained from a **gazetteer**, which is a list of known entity names. For example, the U.S. Social Security Administration provides a list of tens of thousands of

(1) 日文 章魚 怎麼 說?
Japanese octopus how say
How to say octopus in Japanese?

(2) 日 文章 魚 怎麼 說?
Japan essay fish how say

Figure 8.3: An example of tokenization ambiguity in Chinese (Sproat et al., 1996)

given names — more than could be observed in any annotated corpus. Tokens or spans that match an entry in a gazetteer can receive special features; this provides a way to incorporate hand-crafted resources such as name lists in a learning-driven framework.

Neural sequence labeling for NER Current research has emphasized neural sequence labeling, using similar LSTM models to those employed in part-of-speech tagging (Hammerton, 2003; Huang et al., 2015; Lample et al., 2016). The bidirectional LSTM-CRF (Figure 7.4 in § 7.6) does particularly well on this task, due to its ability to model tag-to-tag dependencies. However, Strubell et al. (2017) show that **convolutional neural networks** can be equally accurate, with significant improvement in speed due to the efficiency of implementing ConvNets on **graphics processing units (GPUs)**. The key innovation in this work was the use of **dilated convolution**, which is described in more detail in § 3.4.

8.4 Tokenization

A basic problem for text analysis, first discussed in § 4.3.1, is to break the text into a sequence of discrete tokens. For alphabetic languages such as English, deterministic scripts usually suffice to achieve accurate tokenization. However, in logographic writing systems such as Chinese script, words are typically composed of a small number of characters, without intervening whitespace. The tokenization must be determined by the reader, with the potential for occasional ambiguity, as shown in Figure 8.3. One approach is to match character sequences against a known dictionary (e.g., Sproat et al., 1996), using additional statistical information about word frequency. However, no dictionary is completely comprehensive, and dictionary-based approaches can struggle with such out-of-vocabulary words.

Chinese word segmentation has therefore been approached as a supervised sequence labeling problem. Xue et al. (2003) train a logistic regression classifier to make independent segmentation decisions while moving a sliding window across the document. A set of rules is then used to convert these individual classification decisions into an overall tokenization of the input. However, these individual decisions may be globally suboptimal, motivating a structure prediction approach. Peng et al. (2004) train a conditional random

field to predict labels of START or NONSTART on each character. More recent work has employed neural network architectures. For example, Chen et al. (2015) use an LSTM-CRF architecture, as described in § 7.6: they construct a trellis, in which each tag is scored according to the hidden state of an LSTM, and tag-tag transitions are scored according to learned transition weights. The best-scoring segmentation is then computed by the Viterbi algorithm.

8.5 Code switching

Multilingual speakers and writers do not restrict themselves to a single language. **Code switching** is the phenomenon of switching between languages in speech and text (Auer, 2013; Poplack, 1980). Written code switching has become more common in online social media, as in the following extract from the website of Canadian President Justin Trudeau:[7]

(8.21) *Although everything written on this site est disponible en anglais*
is available in English
and in French, my personal videos seront bilingues
will be bilingual

Accurately analyzing such texts requires first determining which languages are being used. Furthermore, quantitative analysis of code switching can provide insights on the languages themselves and their relative social positions.

Code switching can be viewed as a sequence labeling problem, where the goal is to label each token as a candidate switch point. In the example above, the words *est*, *and*, and *seront* would be labeled as switch points. Solorio and Liu (2008) detect English-Spanish switch points using a supervised classifier, with features that include the word, its part-of-speech in each language (according to a supervised part-of-speech tagger), and the probabilities of the word and part-of-speech in each language. Nguyen and Dogruöz (2013) apply a conditional random field to the problem of detecting code switching between Turkish and Dutch.

Code switching is a special case of the more general problem of word level language identification, which Barman et al. (2014) address in the context of trilingual code switching between Bengali, English, and Hindi. They further observe an even more challenging phenomenon: intra-word code switching, such as the use of English suffixes with Bengali roots. They therefore mark each token as either (1) belonging to one of the three languages; (2) a mix of multiple languages; (3) "universal" (e.g., symbols, numbers, emoticons); or (4) undefined.

[7] As quoted in `http://blogues.lapresse.ca/lagace/2008/09/08/justin-trudeau-really-parfait-bilingue/`, accessed August 21, 2017.

Speaker	Dialogue Act	Utterance
A	YES-NO-QUESTION	*So do you go college right now?*
A	ABANDONED	*Are yo-*
B	YES-ANSWER	*Yeah,*
B	STATEMENT	*It's my last year [laughter].*
A	DECLARATIVE-QUESTION	*You're a, so you're a senior now.*
B	YES-ANSWER	*Yeah,*
B	STATEMENT	*I'm working on my projects trying to graduate [laughter]*
A	APPRECIATION	*Oh, good for you.*
B	BACKCHANNEL	*Yeah.*

Figure 8.4: An example of dialogue act labeling (Stolcke et al., 2000)

8.6 Dialogue acts

The sequence labeling problems that we have discussed so far have been over sequences of word tokens or characters (in the case of tokenization). However, sequence labeling can also be performed over higher-level units, such as **utterances**. **Dialogue acts** are labels over utterances in a dialogue, corresponding roughly to the speaker's intention — the utterance's **illocutionary force** (Austin, 1962). For example, an utterance may state a proposition (*it is not down on any map*), pose a question (*shall we keep chasing this murderous fish?*), or provide a response (*aye aye!*). Stolcke et al. (2000) describe how a set of 42 dialogue acts were annotated for the 1,155 conversations in the Switchboard corpus (Godfrey et al., 1992).[8]

An example is shown in Figure 8.4. The annotation is performed over UTTERANCES, with the possibility of multiple utterances per **conversational turn** (in cases such as interruptions, an utterance may split over multiple turns). Some utterances are clauses (e.g., *So do you go to college right now?*), while others are single words (e.g., *yeah*). Stolcke et al. (2000) report that hidden Markov models (HMMs) achieve 96% accuracy on supervised utterance segmentation. The labels themselves reflect the conversational goals of the speaker: the utterance *yeah* functions as an answer in response to the question *you're a senior now*, but in the final line of the excerpt, it is a **backchannel** (demonstrating comprehension).

For task of dialogue act labeling, Stolcke et al. (2000) apply a hidden Markov model. The probability $\text{p}(\boldsymbol{w}_m \mid y_m)$ must generate the entire sequence of words in the utterance, and it is modeled as a trigram language model (§ 6.1). Stolcke et al. (2000) also account for acoustic features, which capture the **prosody** of each utterance — for example, tonal and rhythmic properties of speech, which can be used to distinguish dialogue acts such

[8]Dialogue act modeling is not restricted to speech; it is relevant in any interactive conversation. For example, Jeong et al. (2009) annotate a more limited set of **speech acts** in a corpus of emails and online forums.

as questions and answers. These features are handled with an additional emission distribution, $p(\boldsymbol{a}_m \mid y_m)$, which is modeled with a probabilistic decision tree (Murphy, 2012). While acoustic features yield small improvements overall, they play an important role in distinguish questions from statements, and agreements from backchannels.

Recurrent neural architectures for dialogue act labeling have been proposed by Kalchbrenner and Blunsom (2013) and Ji et al. (2016), with strong empirical results. Both models are recurrent at the utterance level, so that each complete utterance updates a hidden state. The recurrent-convolutional network of Kalchbrenner and Blunsom (2013) uses convolution to obtain a representation of each individual utterance, while Ji et al. (2016) use a second level of recurrence, over individual words. This enables their method to also function as a language model, giving probabilities over sequences of words in a document.

Exercises

1. Using the Universal Dependencies part-of-speech tags, annotate the following sentences. You may examine the UD tagging guidelines. Tokenization is shown with whitespace. Don't forget about punctuation.

   (8.22) a. I try all things , I achieve what I can .
   b. It was that accursed white whale that razed me .
   c. Better to sleep with a sober cannibal , than a drunk Christian .
   d. Be it what it will , I 'll go to it laughing .

2. Select three short sentences from a recent news article, and annotate them for UD part-of-speech tags. Ask a friend to annotate the same three sentences without looking at your annotations. Compute the rate of agreement, using the Kappa metric defined in § 4.5.2. Then work together to resolve any disagreements.

3. Choose one of the following morphosyntactic attributes: MOOD, TENSE, VOICE. Research the definition of this attribute on the universal dependencies website, `http://universaldependencies.org/u/feat/index.html`. Returning to the examples in the first exercise, annotate all verbs for your chosen attribute. It may be helpful to consult examples from an English-language universal dependencies corpus, available at `https://github.com/UniversalDependencies/UD_English-EWT/tree/master`.

4. Download a dataset annotated for universal dependencies, such as the English Treebank at `https://github.com/UniversalDependencies/UD_English-EWT/tree/master`. This corpus is already segmented into training, development, and test data.

a) First, train a logistic regression or SVM classifier using character suffixes: character n-grams up to length 4. Compute the recall, precision, and F-MEASURE on the development data.

b) Next, augment your classifier using the same character suffixes of the preceding and succeeding tokens. Again, evaluate your classifier on heldout data.

c) Optionally, train a Viterbi-based sequence labeling model, using a toolkit such as CRFSuite (`http://www.chokkan.org/software/crfsuite/`) or your own Viterbi implementation. This is more likely to be helpful for attributes in which agreement is required between adjacent words. For example, many Romance languages require gender and number agreement for determiners, nouns, and adjectives.

5. Provide BIO-style annotation of the named entities (person, place, organization, date, or product) in the following expressions:

(8.23) a. The third mate was Flask, a native of Tisbury, in Martha's Vineyard.

b. Its official Nintendo announced today that they Will release the Nintendo 3DS in north America march 27 (Ritter et al., 2011).

c. Jessica Reif, a media analyst at Merrill Lynch & Co., said, "If they can get up and running with exclusive programming within six months, it doesn't set the venture back that far."[9]

6. Run the examples above through the online version of a named entity recognition tagger, such as the Allen NLP system here: http://demo.allennlp.org/named-entity-recognition. Do the predicted tags match your annotations?

7. Build a whitespace tokenizer for English:

a) Using the NLTK library, download the complete text to the novel *Alice in Wonderland* (Carroll, 1865). Hold out the final 1000 words as a test set.

b) Label each alphanumeric character as a segmentation point, $y_m = 1$ if m is the final character of a token. Label every other character as $y_m = 0$. Then concatenate all the tokens in the training and test sets.Make sure that the number of labels $\{y_m\}_{m=1}^{M}$ is identical to the number of characters $\{c_m\}_{m=1}^{M}$ in your concatenated datasets.

c) Train a logistic regression classifier to predict y_m, using the surrounding characters $\boldsymbol{c}_{m-5:m+5}$ as features. After training the classifier, run it on the test set, using the predicted segmentation points to re-tokenize the text.

[9] From the Message Understanding Conference (MUC-7) dataset (Chinchor and Robinson, 1997).

d) Compute the per-character segmentation accuracy on the test set. You should be able to get at least 88% accuracy.

e) Print out a sample of segmented text from the test set, e.g.

```
Thereareno mice in the air , I ' m afraid , but y oumight cat
   chabat , and that ' svery like a mouse , youknow . But
   docatseat bats , I wonder ?'
```

8. Perform the following extensions to your tokenizer in the previous problem.

a) Train a conditional random field sequence labeler, by incorporating the tag bigrams (y_{m-1}, y_m) as additional features. You may use a structured prediction library such as CRFSuite, or you may want to implement Viterbi yourself. Compare the accuracy with your classification-based approach.

b) Compute the token-level performance: treating the original tokenization as ground truth, compute the number of true positives (tokens that are in both the ground truth and predicted tokenization), false positives (tokens that are in the predicted tokenization but not the ground truth), and false negatives (tokens that are in the ground truth but not the predicted tokenization). Compute the F-measure.

Hint: to match predicted and ground truth tokens, add "anchors" for the start character of each token. The number of true positives is then the size of the intersection of the sets of predicted and ground truth tokens.

c) Apply the same methodology in a more practical setting: tokenization of Chinese, which is written without whitespace. You can find annotated datasets at `http://alias-i.com/lingpipe/demos/tutorial/chineseTokens/read-me.html`.

Chapter 9

Formal language theory

We have now seen methods for learning to label individual words, vectors of word counts, and sequences of words; we will soon proceed to more complex structural transformations. Most of these techniques could apply to counts or sequences from any discrete vocabulary; there is nothing fundamentally linguistic about, say, a hidden Markov model. This raises a basic question that this text has not yet considered: what is a language?

This chapter will take the perspective of **formal language theory**, in which a language is defined as a set of **strings**, each of which is a sequence of elements from a finite alphabet. For interesting languages, there are an infinite number of strings that are in the language, and an infinite number of strings that are not. For example:

- the set of all even-length sequences from the alphabet $\{a, b\}$, e.g., $\{\varnothing, aa, ab, ba, bb, aaaa, aaab, \ldots\}$;
- the set of all sequences from the alphabet $\{a, b\}$ that contain aaa as a substring, e.g., $\{aaa, aaaa, baaa, aaab, \ldots\}$;
- the set of all sequences of English words (drawn from a finite dictionary) that contain at least one verb (a finite subset of the dictionary);
- the PYTHON programming language.

Formal language theory defines classes of languages and their computational properties. Of particular interest is the computational complexity of solving the **membership problem** — determining whether a string is in a language. The chapter will focus on three classes of formal languages: regular, context-free, and "mildly" context-sensitive languages.

A key insight of 20th century linguistics is that formal language theory can be usefully applied to natural languages such as English, by designing formal languages that capture as many properties of the natural language as possible. For many such formalisms, a useful linguistic analysis comes as a byproduct of solving the membership problem. The

membership problem can be generalized to the problems of *scoring* strings for their acceptability (as in language modeling), and of **transducing** one string into another (as in translation).

9.1 Regular languages

If you have written a **regular expression**, then you have defined a **regular language**: a regular language is any language that can be defined by a regular expression. Formally, a regular expression can include the following elements:

- A **literal character** drawn from some finite alphabet Σ.
- The **empty string** ϵ.
- The concatenation of two regular expressions RS, where R and S are both regular expressions. The resulting expression accepts any string that can be decomposed $x = yz$, where y is accepted by R and z is accepted by S.
- The alternation $R \mid S$, where R and S are both regular expressions. The resulting expression accepts a string x if it is accepted by R or it is accepted by S.
- The **Kleene star** R^*, which accepts any string x that can be decomposed into a sequence of strings which are all accepted by R.
- Parenthesization (R), which is used to limit the scope of the concatenation, alternation, and Kleene star operators.

Here are some example regular expressions:

- The set of all even length strings on the alphabet $\{a, b\}$: $((aa)|(ab)|(ba)|(bb))^*$
- The set of all sequences of the alphabet $\{a, b\}$ that contain aaa as a substring: $(a|b)^*aaa(a|b)^*$
- The set of all sequences of English words that contain at least one verb: W^*VW^*, where W is an alternation between all words in the dictionary, and V is an alternation between all verbs ($V \subseteq W$).

This list does not include a regular expression for the Python programming language, because this language is not regular — there is no regular expression that can capture its syntax. We will discuss why towards the end of this section.

Regular languages are **closed** under union, intersection, and concatenation. This means that if two languages L_1 and L_2 are regular, then so are the languages $L_1 \cup L_2$, $L_1 \cap L_2$, and the language of strings that can be decomposed as $s = tu$, with $s \in L_1$ and $t \in L_2$. Regular languages are also closed under negation: if L is regular, then so is the language $\overline{L} = \{s \notin L\}$.

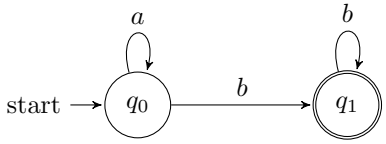


Figure 9.1: State diagram for the finite state acceptor M_1.

9.1.1 Finite state acceptors

A regular expression defines a regular language, but does not give an algorithm for determining whether a string is in the language that it defines. **Finite state automata** are theoretical models of computation on regular languages, which involve transitions between a finite number of states. The most basic type of finite state automaton is the **finite state acceptor (FSA)**, which describes the computation involved in testing if a string is a member of a language. Formally, a finite state acceptor is a tuple $M = (Q, \Sigma, q_0, F, \delta)$, consisting of:

- a finite alphabet Σ of input symbols;
- a finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$;
- a start state $q_0 \in Q$;
- a set of final states $F \subseteq Q$;
- a transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$. The transition function maps from a state and an input symbol (or empty string ϵ) to a *set* of possible resulting states.

A **path** in M is a sequence of transitions, $\pi = t_1, t_2, \ldots, t_N$, where each t_i traverses an arc in the transition function δ. The finite state acceptor M accepts a string ω if there is an accepting path, in which the initial transition t_1 begins at the start state q_0, the final transition t_N terminates in a final state in Q, and the entire input ω is consumed.

Example

Consider the following FSA, M_1.

$$\Sigma = \{a, b\} \quad [9.1]$$
$$Q = \{q_0, q_1\} \quad [9.2]$$
$$F = \{q_1\} \quad [9.3]$$
$$\delta = \{(q_0, a) \rightarrow q_0,\ (q_0, b) \rightarrow q_1,\ (q_1, b) \rightarrow q_1\}. \quad [9.4]$$

This FSA defines a language over an alphabet of two symbols, *a* and *b*. The transition function δ is written as a set of arcs: $(q_0, a) \rightarrow q_0$ says that if the machine is in state

q_0 and reads symbol a, it stays in q_0. Figure 9.1 provides a graphical representation of M_1. Because each pair of initial state and symbol has at most one resulting state, M_1 is **deterministic**: each string ω induces at most one accepting path. Note that there are no transitions for the symbol a in state q_1; if a is encountered in q_1, then the acceptor is stuck, and the input string is rejected.

What strings does M_1 accept? The start state is q_0, and we have to get to q_1, since this is the only final state. Any number of a symbols can be consumed in q_0, but a b symbol is required to transition to q_1. Once there, any number of b symbols can be consumed, but an a symbol cannot. So the regular expression corresponding to the language defined by M_1 is a^*bb^*.

Computational properties of finite state acceptors

The key computational question for finite state acceptors is: how fast can we determine whether a string is accepted? For determistic FSAs, this computation can be performed by Dijkstra's algorithm, with time complexity $\mathcal{O}(V \log V + E)$, where V is the number of vertices in the FSA, and E is the number of edges (Cormen et al., 2009). Non-deterministic FSAs (NFSAs) can include multiple transitions from a given symbol and state. Any NSFA can be converted into a deterministic FSA, but the resulting automaton may have a number of states that is exponential in the number of size of the original NFSA (Mohri et al., 2002).

9.1.2 Morphology as a regular language

Many words have internal structure, such as prefixes and suffixes that shape their meaning. The study of word-internal structure is the domain of **morphology**, of which there are two main types:

- **Derivational morphology** describes the use of affixes to convert a word from one grammatical category to another (e.g., from the noun *grace* to the adjective *graceful*), or to change the meaning of the word (e.g., from *grace* to *disgrace*).
- **Inflectional morphology** describes the addition of details such as gender, number, person, and tense (e.g., the *-ed* suffix for past tense in English).

Morphology is a rich topic in linguistics, deserving of a course in its own right.[1] The focus here will be on the use of finite state automata for morphological analysis. The

[1] A good starting point would be a chapter from a linguistics textbook (e.g., Akmajian et al., 2010; Bender, 2013). A key simplification in this chapter is the focus on affixes at the sole method of derivation and inflection. English makes use of affixes, but also incorporates **apophony**, such as the inflection of *foot* to *feet*. Semitic languages like Arabic and Hebrew feature a template-based system of morphology, in which roots are triples of consonants (e.g., *ktb*), and words are created by adding vowels: ***k**a**t**a**b**a* (Arabic: he wrote), ***k**u**t**u**b*** (books), *ma**k**t**a**b* (desk). For more detail on morphology, see texts from Haspelmath and Sims (2013) and Lieber (2015).

current section deals with derivational morphology; inflectional morphology is discussed in § 9.1.4.

Suppose that we want to write a program that accepts only those words that are constructed in accordance with the rules of English derivational morphology:

(9.1) a. grace, graceful, gracefully, *gracelyful
b. disgrace, *ungrace, disgraceful, disgracefully
c. allure, *allureful, alluring, alluringly
d. fairness, unfair, *disfair, fairly

(Recall that the asterisk indicates that a linguistic example is judged unacceptable by fluent speakers of a language.) These examples cover only a tiny corner of English derivational morphology, but a number of things stand out. The suffix *-ful* converts the nouns *grace* and *disgrace* into adjectives, and the suffix *-ly* converts adjectives into adverbs. These suffixes must be applied in the correct order, as shown by the unacceptability of **gracelyful*. The *-ful* suffix works for only some words, as shown by the use of *alluring* as the adjectival form of *allure*. Other changes are made with prefixes, such as the derivation of *disgrace* from *grace*, which roughly corresponds to a negation; however, *fair* is negated with the *un-* prefix instead. Finally, while the first three examples suggest that the direction of derivation is noun $\rightarrow$ adjective $\rightarrow$ adverb, the example of *fair* suggests that the adjective can also be the base form, with the *-ness* suffix performing the conversion to a noun.

Can we build a computer program that accepts only well-formed English words, and rejects all others? This might at first seem trivial to solve with a brute-force attack: simply make a dictionary of all valid English words. But such an approach fails to account for morphological **productivity** — the applicability of existing morphological rules to new words and names, such as *Trump* to *Trumpy* and *Trumpkin*, and *Clinton* to *Clintonian* and *Clintonite*. We need an approach that represents morphological rules explicitly, and for this we will try a finite state acceptor.

The dictionary approach can be implemented as a finite state acceptor, with the vocabulary Σ equal to the vocabulary of English, and a transition from the start state to the accepting state for each word. But this would of course fail to generalize beyond the original vocabulary, and would not capture anything about the **morphotactic** rules that govern derivations from new words. The first step towards a more general approach is shown in Figure 9.2, which is the state diagram for a finite state acceptor in which the vocabulary consists of **morphemes**, which include **stems** (e.g., *grace, allure*) and **affixes** (e.g., *dis-, -ing, -ly*). This finite state acceptor consists of a set of paths leading away from the start state, with derivational affixes added along the path. Except for q_{neg}, the states on these paths are all final, so the FSA will accept *disgrace*, *disgraceful*, and *disgracefully*, but not *dis-*.

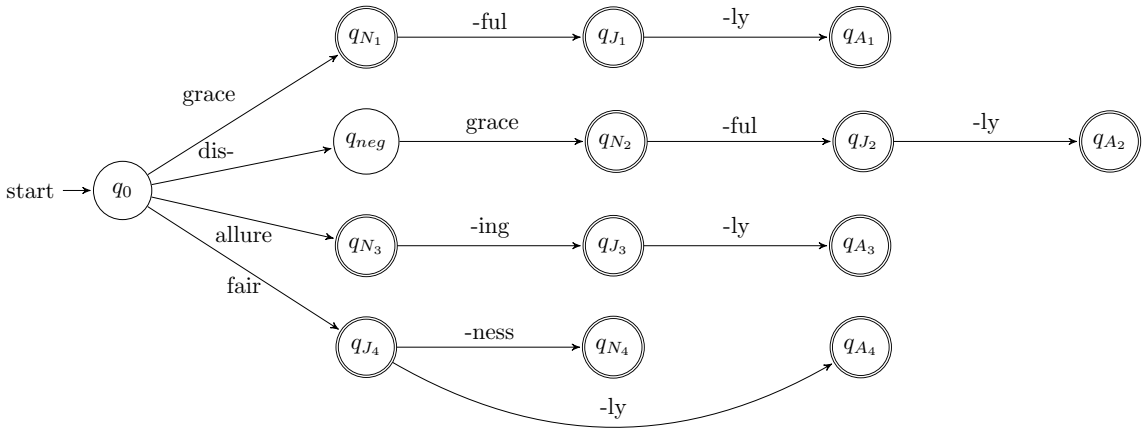


Figure 9.2: A finite state acceptor for a fragment of English derivational morphology. Each path represents possible derivations from a single root form.

This FSA can be **minimized** to the form shown in Figure 9.3, which makes the generality of the finite state approach more apparent. For example, the transition from q_0 to q_{J_2} can be made to accept not only *fair* but any single-morpheme (**monomorphemic**) adjective that takes *-ness* and *-ly* as suffixes. In this way, the finite state acceptor can easily be extended: as new word stems are added to the vocabulary, their derived forms will be accepted automatically. Of course, this FSA would still need to be extended considerably to cover even this small fragment of English morphology. As shown by cases like *music* $\rightarrow$ *musical*, *athlete* $\rightarrow$ *athletic*, English includes several classes of nouns, each with its own rules for derivation.

The FSAs shown in Figure 9.2 and 9.3 accept *allureing*, not *alluring*. This reflects a distinction between morphology — the question of which morphemes to use, and in what order — and **orthography** — the question of how the morphemes are rendered in written language. Just as orthography requires dropping the *e* preceding the *-ing* suffix, **phonology** imposes a related set of constraints on how words are rendered in speech. As we will see soon, these issues can be handled by **finite state!transducers**, which are finite state automata that take inputs and produce outputs.

9.1.3 Weighted finite state acceptors

According to the FSA treatment of morphology, every word is either in or out of the language, with no wiggle room. Perhaps you agree that *musicky* and *fishful* are not valid English words; but if forced to choose, you probably find *a fishful stew* or *a musicky tribute* preferable to *behaving disgracelyful*. Rather than asking whether a word is acceptable, we might like to ask how acceptable it is. Aronoff (1976, page 36) puts it another way:

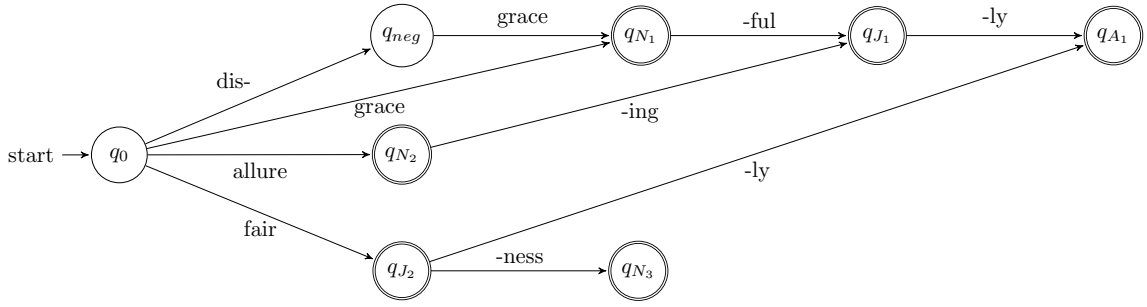


Figure 9.3: Minimization of the finite state acceptor shown in Figure 9.2.

"Though many things are possible in morphology, some are more possible than others." But finite state acceptors give no way to express preferences among technically valid choices.

Weighted finite state acceptors (WFSAs) are generalizations of FSAs, in which each accepting path is assigned a score, computed from the transitions, the initial state, and the final state. Formally, a weighted finite state acceptor $M = (Q, \Sigma, \lambda, \rho, \delta)$ consists of:

- a finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$;
- a finite alphabet Σ of input symbols;
- an initial weight function, $\lambda : Q \rightarrow \mathbb{R}$;
- a final weight function $\rho : Q \rightarrow \mathbb{R}$;
- a transition function $\delta : Q \times \Sigma \times Q \rightarrow \mathbb{R}$.

WFSAs depart from the FSA formalism in three ways: every state can be an initial state, with score $\lambda(q)$; every state can be an accepting state, with score $\rho(q)$; transitions are possible between any pair of states on any input, with a score $\delta(q_i, \omega, q_j)$. Nonetheless, FSAs can be viewed as a special case: for any FSA M we can build an equivalent WFSA by setting $\lambda(q) = \infty$ for all $q \neq q_0$, $\rho(q) = \infty$ for all $q \notin F$, and $\delta(q_i, \omega, q_j) = \infty$ for all transitions $\{(q_1, \omega) \rightarrow q_2\}$ that are not permitted by the transition function of M.

The total score for any path $\pi = t_1, t_2, \ldots, t_N$ is equal to the sum of these scores,

$$d(\pi) = \lambda(\text{from-state}(t_1)) + \sum_{n}^{N} \delta(t_n) + \rho(\text{to-state}(t_N)). \quad [9.5]$$

A **shortest-path algorithm** is used to find the minimum-cost path through a WFSA for string ω, with time complexity $\mathcal{O}(E + V \log V)$, where E is the number of edges and V is the number of vertices (Cormen et al., 2009).[2]

[2]Shortest-path algorithms find the path with the minimum cost. In many cases, the path weights are log

N-gram language models as WFSAs

In n**-gram language models** (see § 6.1), the probability of a sequence of tokens $w_1, w_2, \ldots, w_M$ is modeled as,

$$\mathrm{p}(w_1, \ldots, w_M) \approx \prod_{m=1}^{M} \mathrm{p}_n(w_m \mid w_{m-1}, \ldots, w_{m-n+1}). \quad [9.6]$$

The log probability under an n-gram language model can be modeled in a WFSA. First consider a unigram language model. We need only a single state q_0, with transition scores $\delta(q_0, \omega, q_0) = \log \mathrm{p}_1(\omega)$. The initial and final scores can be set to zero. Then the path score for $w_1, w_2, \ldots, w_M$ is equal to,

$$0 + \sum_{m}^{M} \delta(q_0, w_m, q_0) + 0 = \sum_{m}^{M} \log \mathrm{p}_1(w_m). \quad [9.7]$$

For an n-gram language model with $n > 1$, we need probabilities that condition on the past history. For example, in a bigram language model, the transition weights must represent $\log \mathrm{p}_2(w_m \mid w_{m-1})$. The transition scoring function must somehow "remember" the previous word or words. This can be done by adding more states: to model the bigram probability $\mathrm{p}_2(w_m \mid w_{m-1})$, we need a state for every possible w_{m-1} — a total of V states. The construction indexes each state q_i by a context event $w_{m-1} = i$. The weights are then assigned as follows:

$$\begin{aligned}
\delta(q_i, \omega, q_j) &= \begin{cases} \log \Pr(w_m = j \mid w_{m-1} = i), & \omega = j \\ -\infty, & \omega \neq j \end{cases} \\
\lambda(q_i) &= \log \Pr(w_1 = i \mid w_0 = \square) \\
\rho(q_i) &= \log \Pr(w_{M+1} = \blacksquare \mid w_M = i).
\end{aligned}$$

The transition function is designed to ensure that the context is recorded accurately: we can move to state j on input ω only if $\omega = j$; otherwise, transitioning to state j is forbidden by the weight of $-\infty$. The initial weight function $\lambda(q_i)$ is the log probability of receiving i as the first token, and the final weight function $\rho(q_i)$ is the log probability of receiving an "end-of-string" token after observing $w_M = i$.

*Semiring weighted finite state acceptors

The n-gram language model WFSA is deterministic: each input has exactly one accepting path, for which the WFSA computes a score. In non-deterministic WFSAs, a given input

probabilities, so we want the path with the maximum score, which can be accomplished by making each local score into a *negative* log-probability.

may have multiple accepting paths. In some applications, the score for the input is aggregated across all such paths. Such aggregate scores can be computed by generalizing WFSAs with **semiring notation**, first introduced in § 7.7.3.

Let $d(\pi)$ represent the total score for path $\pi = t_1, t_2, \ldots, t_N$, which is computed as,

$$d(\pi) = \lambda(\text{from-state}(t_1)) \otimes \delta(t_1) \otimes \delta(t_2) \otimes \ldots \otimes \delta(t_N) \otimes \rho(\text{to-state}(t_N)). \quad [9.8]$$

This is a generalization of Equation 9.5 to semiring notation, using the semiring multiplication operator $\otimes$ in place of addition.

Now let $s(\omega)$ represent the total score for all paths $\Pi(\omega)$ that consume input ω,

$$s(\omega) = \bigoplus_{\pi \in \Pi(\omega)} d(\pi). \quad [9.9]$$

Here, semiring addition ($\oplus$) is used to combine the scores of multiple paths.

The generalization to semirings covers a number of useful special cases. In the log-probability semiring, multiplication is defined as $\log \text{p}(x) \otimes \log \text{p}(y) = \log \text{p}(x) + \log \text{p}(y)$, and addition is defined as $\log \text{p}(x) \oplus \log \text{p}(y) = \log(\text{p}(x) + \text{p}(y))$. Thus, $s(\omega)$ represents the log-probability of accepting input ω, marginalizing over all paths $\pi \in \Pi(\omega)$. In the **boolean semiring**, the $\otimes$ operator is logical conjunction, and the $\oplus$ operator is logical disjunction. This reduces to the special case of unweighted finite state acceptors, where the score $s(\omega)$ is a boolean indicating whether there exists any accepting path for ω. In the **tropical semiring**, the $\oplus$ operator is a maximum, so the resulting score is the score of the best-scoring path through the WFSA. The OpenFST toolkit uses semirings and polymorphism to implement general algorithms for weighted finite state automata (Allauzen et al., 2007).

*Interpolated n-gram language models

Recall from § 6.2.3 that an **interpolated n-gram language model** combines the probabilities from multiple n-gram models. For example, an interpolated bigram language model computes the probability,

$$\hat{\text{p}}(w_m \mid w_{m-1}) = \lambda_1 \text{p}_1(w_m) + \lambda_2 \text{p}_2(w_m \mid w_{m-1}), \quad [9.10]$$

with $\hat{\text{p}}$ indicating the interpolated probability, p_2 indicating the bigram probability, and p_1 indicating the unigram probability. Setting $\lambda_2 = (1 - \lambda_1)$ ensures that the probabilities sum to one.

Interpolated bigram language models can be implemented using a non-deterministic WFSA (Knight and May, 2009). The basic idea is shown in Figure 9.4. In an interpolated bigram language model, there is one state for each element in the vocabulary — in this

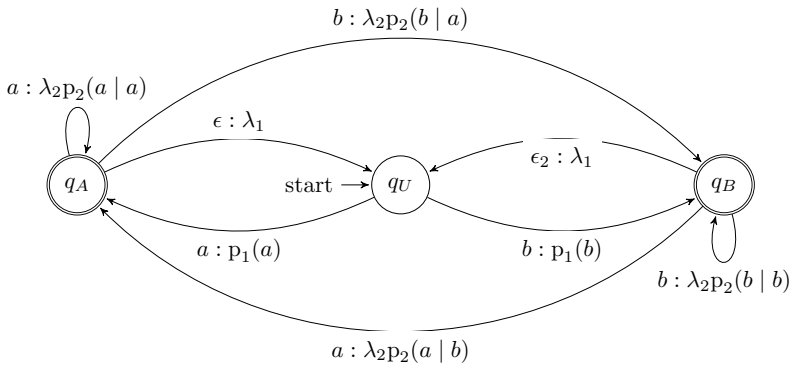


Figure 9.4: WFSA implementing an interpolated bigram/unigram language model, on the alphabet $\Sigma = \{a, b\}$. For simplicity, the WFSA is contrained to force the first token to be generated from the unigram model, and does not model the emission of the end-of-sequence token.

case, the states q_A and q_B — which are capture the contextual conditioning in the bigram probabilities. To model unigram probabilities, there is an additional state q_U, which "forgets" the context. Transitions out of q_U involve unigram probabilities, $\mathrm{p}_1(a)$ and $\mathrm{p}_2(b)$; transitions into q_U emit the empty symbol ϵ, and have probability λ_1, reflecting the interpolation weight for the unigram model. The interpolation weight for the bigram model is included in the weight of the transition $q_A \rightarrow q_B$.

The epsilon transitions into q_U make this WFSA non-deterministic. Consider the score for the sequence (a, b, b). The initial state is q_U, so the symbol a is generated with score $\mathrm{p}_1(a)$[3] Next, we can generate b from the unigram model by taking the transition $q_A \rightarrow q_B$, with score $\lambda_2 \mathrm{p}_2(b \mid a)$. Alternatively, we can take a transition back to q_U with score λ_1, and then emit b from the unigram model with score $\mathrm{p}_1(b)$. To generate the final b token, we face the same choice: emit it directly from the self-transition to q_B, or transition to q_U first.

The total score for the sequence (a, b, b) is the semiring sum over all accepting paths,

$$\begin{aligned} s(a,b,b) = &\big(\mathrm{p}_1(a) \otimes \lambda_2 \mathrm{p}_2(b \mid a) \otimes \lambda_2 \mathrm{p}(b \mid b)\big) \\ &\oplus \big(\mathrm{p}_1(a) \otimes \lambda_1 \otimes \mathrm{p}_1(b) \otimes \lambda_2 \mathrm{p}(b \mid b)\big) \\ &\oplus \big(\mathrm{p}_1(a) \otimes \lambda_2 \mathrm{p}_2(b \mid a) \otimes \mathrm{p}_1(b) \otimes \mathrm{p}_1(b)\big) \\ &\oplus \big(\mathrm{p}_1(a) \otimes \lambda_1 \otimes \mathrm{p}_1(b) \otimes \mathrm{p}_1(b) \otimes \mathrm{p}_1(b)\big) \,. \end{aligned} \quad [9.11]$$

Each line in Equation 9.11 represents the probability of a specific path through the WFSA. In the probability semiring, $\otimes$ is multiplication, so that each path is the product of each

[3] We could model the sequence-initial bigram probability $\mathrm{p}_2(a \mid \square)$, but for simplicity the WFSA does not admit this possibility, which would require another state.

transition weight, which are themselves probabilities. The $\oplus$ operator is addition, so that the total score is the sum of the scores (probabilities) for each path. This corresponds to the probability under the interpolated bigram language model.

9.1.4 Finite state transducers

Finite state acceptors can determine whether a string is in a regular language, and weighted finite state acceptors can compute a score for every string over a given alphabet. **Finite state transducers** (FSTs) extend the formalism further, by adding an output symbol to each transition. Formally, a finite state transducer is a tuple $T = (Q, \Sigma, \Omega, \lambda, \rho, \delta)$, with Ω representing an output vocabulary and the transition function $\delta : Q \times (\Sigma \cup \epsilon) \times (\Omega \cup \epsilon) \times Q \rightarrow \mathbb{R}$ mapping from states, input symbols, and output symbols to states. The remaining elements (Q, Σ, λ, ρ) are identical to their definition in weighted finite state acceptors (§ 9.1.3). Thus, each path through the FST T transduces the input string into an output.

String edit distance

The **edit distance** between two strings s and t is a measure of how many operations are required to transform one string into another. There are several ways to compute edit distance, but one of the most popular is the Levenshtein edit distance, which counts the minimum number of insertions, deletions, and substitutions. This can be computed by a one-state weighted finite state transducer, in which the input and output alphabets are identical. For simplicity, consider the alphabet $\Sigma = \Omega = \{a, b\}$. The edit distance can be computed by a one-state transducer with the following transitions,

$$\delta(q, a, a, q) = \delta(q, b, b, q) = 0 \quad [9.12]$$
$$\delta(q, a, b, q) = \delta(q, b, a, q) = 1 \quad [9.13]$$
$$\delta(q, a, \epsilon, q) = \delta(q, b, \epsilon, q) = 1 \quad [9.14]$$
$$\delta(q, \epsilon, a, q) = \delta(q, \epsilon, b, q) = 1. \quad [9.15]$$

The state diagram is shown in Figure 9.5.

For a given string pair, there are multiple paths through the transducer: the best-scoring path from *dessert* to *desert* involves a single deletion, for a total score of 1; the worst-scoring path involves seven deletions and six additions, for a score of 13.

The Porter stemmer

The Porter (1980) stemming algorithm is a "lexicon-free" algorithm for stripping suffixes from English words, using a sequence of character-level rules. Each rule can be described

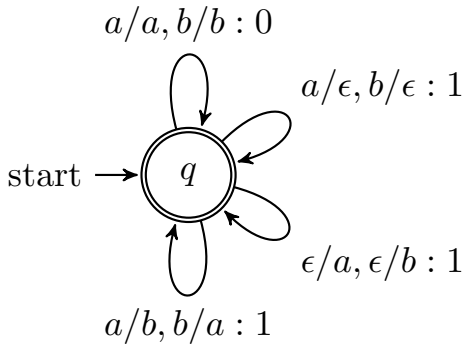


Figure 9.5: State diagram for the Levenshtein edit distance finite state transducer. The label $x/y : c$ indicates a cost of c for a transition with input x and output y.

by an unweighted finite state transducer. The first rule is:

$$\textit{-sses} \rightarrow \textit{-ss} \quad \text{e.g., } \textit{dresses} \rightarrow \textit{dress} \tag{9.16}$$
$$\textit{-ies} \rightarrow \textit{-i} \quad \text{e.g., } \textit{parties} \rightarrow \textit{parti} \tag{9.17}$$
$$\textit{-ss} \rightarrow \textit{-ss} \quad \text{e.g., } \textit{dress} \rightarrow \textit{dress} \tag{9.18}$$
$$\textit{-s} \rightarrow \epsilon \quad \text{e.g., } \textit{cats} \rightarrow \textit{cat} \tag{9.19}$$

The final two lines appear to conflict; they are meant to be interpreted as an instruction to remove a terminal *-s* unless it is part of an *-ss* ending. A state diagram to handle just these final two lines is shown in Figure 9.6. Make sure you understand how this finite state transducer handles *cats*, *steps*, *bass*, and *basses*.

Inflectional morphology

In **inflectional morphology**, word **lemmas** are modified to add grammatical information such as tense, number, and case. For example, many English nouns are pluralized by the suffix *-s*, and many verbs are converted to past tense by the suffix *-ed*. English's inflectional morphology is considerably simpler than many of the world's languages. For example, Romance languages (derived from Latin) feature complex systems of verb suffixes which must agree with the person and number of the verb, as shown in Table 9.1.

The task of morphological analysis is to read a form like *canto*, and output an analysis like CANTAR+VERB+PRESIND+1P+SING, where +PRESIND describes the tense as present indicative, +1P indicates the first-person, and +SING indicates the singular number. The task of morphological generation is the reverse, going from CANTAR+VERB+PRESIND+1P+SING to *canto*. Finite state transducers are an attractive solution, because they can solve both problems with a single model (Beesley and Karttunen, 2003). As an example, Figure 9.7 shows a fragment of a finite state transducer for Spanish inflectional morphology. The

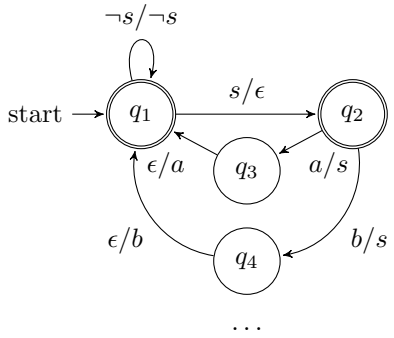


Figure 9.6: State diagram for final two lines of step 1a of the Porter stemming diagram. States q_3 and q_4 "remember" the observations a and b respectively; the ellipsis ... represents additional states for each symbol in the input alphabet. The notation $\neg s/\neg s$ is not part of the FST formalism; it is a shorthand to indicate a set of self-transition arcs for every input/output symbol except s.

infinitive	cantar (to sing)	comer (to eat)	vivir (to live)
yo (1st singular)	canto	como	vivo
tu (2nd singular)	cantas	comes	vives
él, ella, usted (3rd singular)	canta	come	vive
nosotros (1st plural)	cantamos	comemos	vivimos
vosotros (2nd plural, informal)	cantáis	coméis	vivís
ellos, ellas (3rd plural); ustedes (2nd plural)	cantan	comen	viven

Table 9.1: Spanish verb inflections for the present indicative tense. Each row represents a person and number, and each column is a regular example from a class of verbs, as indicated by the ending of the infinitive form.

input vocabulary Σ corresponds to the set of letters used in Spanish spelling, and the output vocabulary Ω corresponds to these same letters, plus the vocabulary of morphological features (e.g., +SING, +VERB). In Figure 9.7, there are two paths that take *canto* as input, corresponding to the verb and noun meanings; the choice between these paths could be guided by a part-of-speech tagger. By **inversion**, the inputs and outputs for each transition are switched, resulting in a finite state generator, capable of producing the correct **surface form** for any morphological analysis.

Finite state morphological analyzers and other unweighted transducers can be designed by hand. The designer's goal is to avoid **overgeneration** — accepting strings or making transductions that are not valid in the language — as well as **undergeneration**

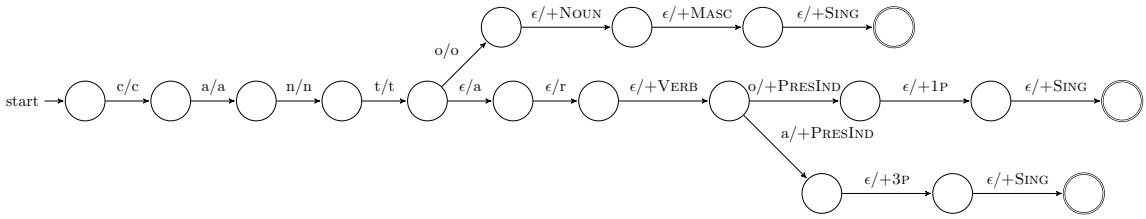


Figure 9.7: Fragment of a finite state transducer for Spanish morphology. There are two accepting paths for the input *canto*: *canto*+NOUN+MASC+SING (masculine singular noun, meaning a song), and *cantar*+VERB+PRESIND+1P+SING (I sing). There is also an accepting path for *canta*, with output *cantar*+VERB+PRESIND+3P+SING (he/she sings).

— failing to accept strings or transductions that are valid. For example, a pluralization transducer that does not accept *foot/feet* would undergenerate. Suppose we "fix" the transducer to accept this example, but as a side effect, it now accepts *boot/beet*; the transducer would then be said to overgenerate. If a transducer accepts *foot/foots* but not *foot/feet*, then it simultaneously overgenerates and undergenerates.

Finite state composition

Designing finite state transducers to capture the full range of morphological phenomena in any real language is a huge task. Modularization is a classic computer science approach for this situation: decompose a large and unwieldly problem into a set of subproblems, each of which will hopefully have a concise solution. Finite state automata can be modularized through **composition**: feeding the output of one transducer T_1 as the input to another transducer T_2, written $T_2 \circ T_1$. Formally, if there exists some y such that $(x, y) \in T_1$ (meaning that T_1 produces output y on input x), and $(y, z) \in T_2$, then $(x, z) \in (T_2 \circ T_1)$. Because finite state transducers are closed under composition, there is guaranteed to be a single finite state transducer that $T_3 = T_2 \circ T_1$, which can be constructed as a machine with one state for each pair of states in T_1 and T_2 (Mohri et al., 2002).

Example: Morphology and orthography In English morphology, the suffix *-ed* is added to signal the past tense for many verbs: *cook*→*cooked*, *want*→*wanted*, etc. However, English **orthography** dictates that this process cannot produce a spelling with consecutive *e*'s, so that *bake*→*baked*, not *bakeed*. A modular solution is to build separate transducers for morphology and orthography. The morphological transducer T_M transduces from *bake*+PAST to *bake*+*ed*, with the + symbol indicating a segment boundary. The input alphabet of T_M includes the lexicon of words and the set of morphological features; the output alphabet includes the characters *a*-*z* and the + boundary marker. Next, an orthographic transducer T_O is responsible for the transductions *cook*+*ed* → *cooked*, and *bake*+*ed* → *baked*. The input alphabet of T_O must be the same as the output alphabet for T_M, and the output alphabet

is simply the characters *a*-*z*. The composed transducer $(T_O \circ T_M)$ then transduces from *bake*+PAST to the spelling *baked*. The design of T_O is left as an exercise.

Example: Hidden Markov models Hidden Markov models (chapter 7) can be viewed as weighted finite state transducers, and they can be constructed by transduction. Recall that a hidden Markov model defines a joint probability over words and tags, $\text{p}(\boldsymbol{w}, \boldsymbol{y})$, which can be computed as a path through a **trellis** structure. This trellis is itself a weighted finite state acceptor, with edges between all adjacent nodes $q_{m-1,i} \rightarrow q_{m,j}$ on input $Y_m = j$. The edge weights are log-probabilities,

$$\begin{aligned}\delta(q_{m-1,i}, Y_m = j, q_{m,j}) &= \log \text{p}(w_m, Y_m = j \mid Y_{m-i} = j) && [9.20]\\ &= \log \text{p}(w_m \mid Y_m = j) + \log \Pr(Y_m = j \mid Y_{m-1} = i). && [9.21]\end{aligned}$$

Because there is only one possible transition for each tag Y_m, this WFSA is deterministic. The score for any tag sequence $\{y_m\}_{m=1}^M$ is the sum of these log-probabilities, corresponding to the total log probability $\log \text{p}(\boldsymbol{w}, \boldsymbol{y})$. Furthermore, the trellis can be constructed by the composition of simpler FSTs.

- First, construct a "transition" transducer to represent a bigram probability model over tag sequences, T_T. This transducer is almost identical to the n-gram language model acceptor in § 9.1.3: there is one state for each tag, and the edge weights equal to the transition log-probabilities, $\delta(q_i, j, j, q_j) = \log \Pr(Y_m = j \mid Y_{m-1} = i)$. Note that T_T is a transducer, with identical input and output at each arc; this makes it possible to compose T_T with other transducers.
- Next, construct an "emission" transducer to represent the probability of words given tags, T_E. This transducer has only a single state, with arcs for each word/tag pair, $\delta(q_0, i, j, q_0) = \log \Pr(W_m = j \mid Y_m = i)$. The input vocabulary is the set of all tags, and the output vocabulary is the set of all words.
- The composition $T_E \circ T_T$ is a finite state transducer with one state per tag, as shown in Figure 9.8. Each state has $V \times K$ outgoing edges, representing transitions to each of the K other states, with outputs for each of the V words in the vocabulary. The weights for these edges are equal to,

$$\delta(q_i, Y_m = j, w_m, q_j) = \log \text{p}(w_m, Y_m = j \mid Y_{m-1} = i). \quad [9.22]$$

- The trellis is a structure with $M \times K$ nodes, for each of the M words to be tagged and each of the K tags in the tagset. It can be built by composition of $(T_E \circ T_T)$ against an unweighted **chain FSA** $M_A(\boldsymbol{w})$ that is specially constructed to accept only a given input $w_1, w_2, \ldots, w_M$, shown in Figure 9.9. The trellis for input $\boldsymbol{w}$ is built from the composition $M_A(\boldsymbol{w}) \circ (T_E \circ T_T)$. Composing with the unweighted $M_A(\boldsymbol{w})$ does not affect the edge weights from $(T_E \circ T_T)$, but it selects the subset of paths that generate the word sequence $\boldsymbol{w}$.

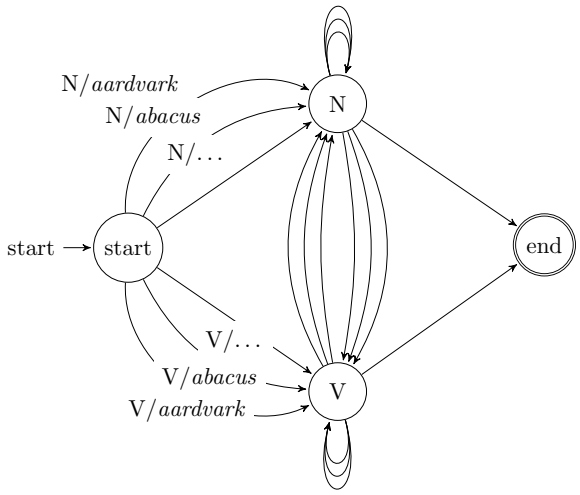


Figure 9.8: Finite state transducer for hidden Markov models, with a small tagset of **n**ouns and **v**erbs. For each pair of tags (including self-loops), there is an edge for every word in the vocabulary. For simplicity, input and output are only shown for the edges from the start state. Weights are also omitted from the diagram; for each edge from q_i to q_j, the weight is equal to $\log \mathrm{p}(w_m, Y_m = j \mid Y_{m-1} = i)$, except for edges to the end state, which are equal to $\log \Pr(Y_m = \blacklozenge \mid Y_{m-1} = i)$.

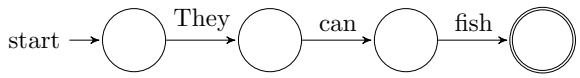


Figure 9.9: Chain finite state acceptor for the input *They can fish*.

9.1.5 *Learning weighted finite state automata

In generative models such as n-gram language models and hidden Markov models, the edge weights correspond to log probabilities, which can be obtained from relative frequency estimation. However, in other cases, we wish to learn the edge weights from input/output pairs. This is difficult in non-deterministic finite state automata, because we do not observe the specific arcs that are traversed in accepting the input, or in transducing from input to output. The path through the automaton is a **latent variable**.

Chapter 5 presented one method for learning with latent variables: expectation maximization (EM). This involves computing a distribution $q(\cdot)$ over the latent variable, and iterating between updates to this distribution and updates to the parameters — in this case, the arc weights. The **forward-backward algorithm** (§ 7.5.3) describes a dynamic program for computing a distribution over arcs in the trellis structure of a hidden Markov

model, but this is a special case of the more general problem for finite state automata. Eisner (2002) describes an **expectation semiring**, which enables the expected number of transitions across each arc to be computed through a semiring shortest-path algorithm. Alternative approaches for generative models include Markov Chain Monte Carlo (Chiang et al., 2010) and spectral learning (Balle et al., 2011).

Further afield, we can take a perceptron-style approach, with each arc corresponding to a feature. The classic perceptron update would update the weights by subtracting the difference between the feature vector corresponding to the predicted path and the feature vector corresponding to the correct path. Since the path is not observed, we resort to a **latent variable perceptron**. The model is described formally in § 12.4, but the basic idea is to compute an update from the difference between the features from the predicted path and the features for the best-scoring path that generates the correct output.

9.2 Context-free languages

Beyond the class of regular languages lie the context-free languages. An example of a language that is context-free but not finite state is the set of arithmetic expressions with balanced parentheses. Intuitively, to accept only strings in this language, an FSA would have to "count" the number of left parentheses, and make sure that they are balanced against the number of right parentheses. An arithmetic expression can be arbitrarily long, yet by definition an FSA has a finite number of states. Thus, for any FSA, there will be a string with too many parentheses to count. More formally, the **pumping lemma** is a proof technique for showing that languages are not regular. It is typically demonstrated for the simpler case $a^n b^n$, the language of strings containing a sequence of a's, and then an equal-length sequence of b's.[4]

There are at least two arguments for the relevance of non-regular formal languages to linguistics. First, there are natural language phenomena that are argued to be isomorphic to $a^n b^n$. For English, the classic example is **center embedding**, shown in Figure 9.10. The initial expression *the dog* specifies a single dog. Embedding this expression into *the cat ___ chased* specifies a particular cat — the one chased by the dog. This cat can then be embedded again to specify a goat, in the less felicitous but arguably grammatical expression, *the goat the cat the dog chased kissed*, which refers to the goat who was kissed by the cat which was chased by the dog. Chomsky (1957) argues that to be grammatical, a center-embedded construction must be balanced: if it contains n noun phrases (e.g., *the cat*), they must be followed by exactly $n - 1$ verbs. An FSA that could recognize such expressions would also be capable of recognizing the language $a^n b^n$. Because we can prove that no FSA exists for $a^n b^n$, no FSA can exist for center embedded constructions either. En-

[4] Details of the proof can be found in an introductory computer science theory textbook (e.g., Sipser, 2012).

		the dog		
	the cat	the dog	chased	
the goat	the cat	the dog	chased	kissed
		...		

Figure 9.10: Three levels of center embedding

glish includes center embedding, and so the argument goes, English grammar as a whole cannot be regular.[5]

A more practical argument for moving beyond regular languages is modularity. Many linguistic phenomena — especially in syntax — involve constraints that apply at long distance. Consider the problem of determiner-noun number agreement in English: we can say *the coffee* and *these coffees*, but not **these coffee*. By itself, this is easy enough to model in an FSA. However, fairly complex modifying expressions can be inserted between the determiner and the noun:

(9.2) a. the burnt coffee
b. the badly-ground coffee
c. the burnt and badly-ground Italian coffee
d. these burnt and badly-ground Italian coffees
e. * these burnt and badly-ground Italian coffee

Again, an FSA can be designed to accept modifying expressions such as *burnt and badly-ground Italian*. Let's call this FSA F_M. To reject the final example, a finite state acceptor must somehow "remember" that the determiner was plural when it reaches the noun *coffee* at the end of the expression. The only way to do this is to make two identical copies of F_M: one for singular determiners, and one for plurals. While this is possible in the finite state framework, it is inconvenient — especially in languages where more than one attribute of the noun is marked by the determiner. **Context-free languages** facilitate modularity across such long-range dependencies.

9.2.1 Context-free grammars

Context-free languages are specified by **context-free grammars** (CFGs), which are tuples (N, Σ, R, S) consisting of:

[5]The claim that arbitrarily deep center-embedded expressions are grammatical has drawn skepticism. Corpus evidence shows that embeddings of depth greater than two are exceedingly rare (Karlsson, 2007), and that embeddings of depth greater than three are completely unattested. If center-embedding is capped at some finite depth, then it is regular.

$$\begin{aligned}
\text{S} &\rightarrow \text{S OP S} \mid \text{NUM} \\
\text{OP} &\rightarrow + \mid - \mid \times \mid \div \\
\text{NUM} &\rightarrow \text{NUM DIGIT} \mid \text{DIGIT} \\
\text{DIGIT} &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9}
\end{aligned}$$

Figure 9.11: A context-free grammar for arithmetic expressions

- a finite set of **non-terminals** N;
- a finite alphabet Σ of **terminal symbols**;
- a set of **production rules** R, each of the form $A \rightarrow \beta$, where $A \in N$ and $\beta \in (\Sigma \cup N)^*$;
- a designated start symbol S.

In the production rule $A \rightarrow \beta$, the left-hand side (LHS) A must be a non-terminal; the right-hand side (RHS) can be a sequence of terminals or non-terminals, $\{n, \sigma\}^*, n \in N, \sigma \in \Sigma$. A non-terminal can appear on the left-hand side of many production rules. A non-terminal can appear on both the left-hand side and the right-hand side; this is a **recursive production**, and is analogous to self-loops in finite state automata. The name "context-free" is based on the property that the production rule depends only on the LHS, and not on its ancestors or neighbors; this is analogous to Markov property of finite state automata, in which the behavior at each step depends only on the current state, and not on the path by which that state was reached.

A **derivation** τ is a sequence of steps from the start symbol S to a surface string $\boldsymbol{w} \in \Sigma^*$, which is the **yield** of the derivation. A string $\boldsymbol{w}$ is in a context-free language if there is some derivation from S yielding $\boldsymbol{w}$. **Parsing** is the problem of finding a derivation for a string in a grammar. Algorithms for parsing are described in chapter 10.

Like regular expressions, context-free grammars define the language but not the computation necessary to recognize it. The context-free analogues to finite state acceptors are **pushdown automata**, a theoretical model of computation in which input symbols can be pushed onto a stack with potentially infinite depth. For more details, see Sipser (2012).

Example

Figure 9.11 shows a context-free grammar for arithmetic expressions such as $1 + 2 \div 3 - 4$. In this grammar, the terminal symbols include the digits $\{\texttt{1}, \texttt{2}, \ldots, \texttt{9}\}$ and the operators $\{+, -, \times, \div\}$. The rules include the $\mid$ symbol, a notational convenience that makes it possible to specify multiple right-hand sides on a single line: the statement $A \rightarrow x \mid y$

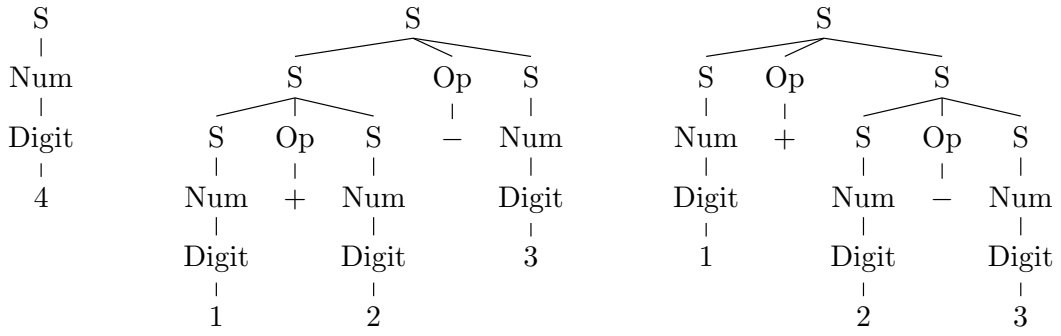


Figure 9.12: Some example derivations from the arithmetic grammar in Figure 9.11

defines *two* productions, $A \rightarrow x$ and $A \rightarrow y$. This grammar is recursive: the non-termals S and NUM can produce themselves.

Derivations are typically shown as trees, with production rules applied from the top to the bottom. The tree on the left in Figure 9.12 describes the derivation of a single digit, through the sequence of productions S $\rightarrow$ NUM $\rightarrow$ DIGIT $\rightarrow$ 4 (these are all **unary productions**, because the right-hand side contains a single element). The other two trees in Figure 9.12 show alternative derivations of the string `1 + 2 - 3`. The existence of multiple derivations for a string indicates that the grammar is **ambiguous**.

Context-free derivations can also be written out according to the pre-order tree traversal.[6] For the two derivations of `1 + 2 - 3` in Figure 9.12, the notation is:

(S (S (S (Num (Digit 1))) (Op +) (S (Num (Digit 2)))) (Op -) (S (Num (Digit 3)))) [9.23]

(S (S (Num (Digit 1))) (Op +) (S (Num (Digit 2)) (Op -) (S (Num (Digit 3))))). [9.24]

Grammar equivalence and Chomsky Normal Form

A single context-free language can be expressed by more than one context-free grammar. For example, the following two grammars both define the language $a^n b^n$ for $n > 0$.

$$S \rightarrow aSb \mid ab$$
$$S \rightarrow aSb \mid aabb \mid ab$$

Two grammars are **weakly equivalent** if they generate the same strings. Two grammars are **strongly equivalent** if they generate the same strings via the same derivations. The grammars above are only weakly equivalent.

[6] This is a depth-first left-to-right search that prints each node the first time it is encountered (Cormen et al., 2009, chapter 12).

In **Chomsky Normal Form (CNF)**, the right-hand side of every production includes either two non-terminals, or a single terminal symbol:

$$A \rightarrow BC$$
$$A \rightarrow a$$

All CFGs can be converted into a CNF grammar that is weakly equivalent. To convert a grammar into CNF, we first address productions that have more than two non-terminals on the RHS by creating new "dummy" non-terminals. For example, if we have the production,

$$\text{W} \rightarrow \text{X Y Z}, \quad [9.25]$$

it is replaced with two productions,

$$\text{W} \rightarrow \text{X W}\backslash\text{X} \quad [9.26]$$
$$\text{W}\backslash\text{X} \rightarrow \text{Y Z}. \quad [9.27]$$

In these productions, W\X is a new dummy non-terminal. This transformation **binarizes** the grammar, which is critical for efficient bottom-up parsing, as we will see in chapter 10. Productions whose right-hand side contains a mix of terminal and non-terminal symbols can be replaced in a similar fashion.

Unary non-terminal productions $A \rightarrow B$ are replaced as follows: for each production $B \rightarrow \alpha$ in the grammar, add a new production $A \rightarrow \alpha$. For example, in the grammar described in Figure 9.11, we would replace NUM $\rightarrow$ DIGIT with NUM $\rightarrow$ `1` | `2` | ... | `9`. However, we keep the production NUM $\rightarrow$ NUM DIGIT, which is a valid binary production.

9.2.2 Natural language syntax as a context-free language

Context-free grammars can be used to represent **syntax**, which is the set of rules that determine whether an utterance is judged to be grammatical. If this representation were perfectly faithful, then a natural language such as English could be transformed into a formal language, consisting of exactly the (infinite) set of strings that would be judged to be grammatical by a fluent English speaker. We could then build parsing software that would automatically determine if a given utterance were grammatical.[7]

Contemporary theories generally do *not* consider natural languages to be context-free (see § 9.3), yet context-free grammars are widely used in natural language parsing. The reason is that context-free representations strike a good balance: they cover a broad range of syntactic phenomena, and they can be parsed efficiently. This section therefore describes how to handle a core fragment of English syntax in context-free form, following

[7] To move beyond this cursory treatment of syntax, consult the short introductory manuscript by Bender (2013), or the longer text by Akmajian et al. (2010).

the conventions of the **Penn Treebank** (PTB; Marcus et al., 1993), a large-scale annotation of English language syntax. The generalization to "mildly" context-sensitive languages is discussed in § 9.3.

The Penn Treebank annotation is a **phrase-structure grammar** of English. This means that sentences are broken down into **constituents**, which are contiguous sequences of words that function as coherent units for the purpose of linguistic analysis. Constituents generally have a few key properties:

Movement. Constituents can often be moved around sentences as units.

(9.3) a. Abigail gave (her brother) (a fish).
b. Abigail gave (a fish) to (her brother).

In contrast, *gave her* and *brother a* cannot easily be moved while preserving grammaticality.

Substitution. Constituents can be substituted by other phrases of the same type.

(9.4) a. Max thanked (his older sister).
b. Max thanked (her).

In contrast, substitution is not possible for other contiguous units like *Max thanked* and *thanked his*.

Coordination. Coordinators like *and* and *or* can conjoin constituents.

(9.5) a. (Abigail) and (her younger brother) bought a fish.
b. Abigail (bought a fish) and (gave it to Max).
c. Abigail (bought) and (greedily ate) a fish.

Units like *brother bought* and *bought a* cannot easily be coordinated.

These examples argue for units such as *her brother* and *bought a fish* to be treated as constituents. Other sequences of words in these examples, such as *Abigail gave* and *brother a fish*, cannot be moved, substituted, and coordinated in these ways. In phrase-structure grammar, constituents are nested, so that *the senator from New Jersey* contains the constituent *from New Jersey*, which in turn contains *New Jersey*. The sentence itself is the maximal constituent; each word is a minimal constituent, derived from a unary production from a part-of-speech tag. Between part-of-speech tags and sentences are **phrases**. In phrase-structure grammar, phrases have a type that is usually determined by their **head word**: for example, a **noun phrase** corresponds to a noun and the group of words that

modify it, such as *her younger **brother***; a **verb phrase** includes the verb and its modifiers, such as ***bought** a fish* and *greedily **ate** it*.

In context-free grammars, each phrase type is a non-terminal, and each constituent is the substring that the non-terminal yields. Grammar design involves choosing the right set of non-terminals. Fine-grained non-terminals make it possible to represent more fine-grained linguistic phenomena. For example, by distinguishing singular and plural noun phrases, it is possible to have a grammar of English that generates only sentences that obey subject-verb agreement. However, enforcing subject-verb agreement is considerably more complicated in languages like Spanish, where the verb must agree in both person and number with subject. In general, grammar designers must trade off between **over-generation** — a grammar that permits ungrammatical sentences — and **undergeneration** — a grammar that fails to generate grammatical sentences. Furthermore, if the grammar is to support manual annotation of syntactic structure, it must be simple enough to annotate efficiently.

9.2.3 A phrase-structure grammar for English

To better understand how phrase-structure grammar works, let's consider the specific case of the Penn Treebank grammar of English. The main phrase categories in the Penn Treebank (PTB) are based on the main part-of-speech classes: noun phrase (NP), verb phrase (VP), prepositional phrase (PP), adjectival phrase (ADJP), and adverbial phrase (ADVP). The top-level category is S, which conveniently stands in for both "sentence" and the "start" symbol. **Complement clauses** (e.g., *I take the good old fashioned ground **that the whale is a fish***) are represented by the non-terminal SBAR. The terminal symbols in the grammar are individual words, which are generated from unary productions from part-of-speech tags (the PTB tagset is described in § 8.1).

This section describes some of the most common productions from the major phrase-level categories, explaining how to generate individual tag sequences. The production rules are approached in a "theory-driven" manner: first the syntactic properties of each phrase type are described, and then some of the necessary production rules are listed. But it is important to keep in mind that the Penn Treebank was produced in a "data-driven" manner. After the set of non-terminals was specified, annotators were free to analyze each sentence in whatever way seemed most linguistically accurate, subject to some high-level guidelines. The grammar of the Penn Treebank is simply the set of productions that were required to analyze the several million words of the corpus. By design, the grammar overgenerates — it does not exclude ungrammatical sentences. Furthermore, while the productions shown here cover some of the most common cases, they are only a small fraction of the several thousand different types of productions in the Penn Treebank.

Sentences

The most common production rule for sentences is,

$$\text{S} \rightarrow \text{NP VP} \quad [9.28]$$

which accounts for simple sentences like *Abigail ate the kimchi* — as we will see, the direct object *the kimchi* is part of the verb phrase. But there are more complex forms of sentences as well:

$$\text{S} \rightarrow \text{ADVP NP VP} \quad \textit{Unfortunately Abigail ate the kimchi.} \quad [9.29]$$

$$\text{S} \rightarrow \text{S Cc S} \quad \textit{Abigail ate the kimchi and Max had a burger.} \quad [9.30]$$

$$\text{S} \rightarrow \text{VP} \quad \textit{Eat the kimchi.} \quad [9.31]$$

where ADVP is an adverbial phrase (e.g., *unfortunately*, *very unfortunately*) and Cc is a coordinating conjunction (e.g., *and*, *but*).[8]

Noun phrases

Noun phrases refer to entities, real or imaginary, physical or abstract: *Asha*, *the steamed dumpling*, *parts and labor*, *nobody*, *the whiteness of the whale*, and *the rise of revolutionary syndicalism in the early twentieth century.* Noun phrase productions include "bare" nouns, which may optionally follow determiners, as well as pronouns:

$$\text{NP} \rightarrow \text{Nn} \mid \text{Nns} \mid \text{Nnp} \mid \text{Prp} \quad [9.32]$$

$$\text{NP} \rightarrow \text{Det Nn} \mid \text{Det Nns} \mid \text{Det Nnp} \quad [9.33]$$

The tags Nn, Nns, and Nnp refer to singular, plural, and proper nouns; Prp refers to personal pronouns, and Det refers to determiners. The grammar also contains terminal productions from each of these tags, e.g., Prp $\rightarrow$ *I* | *you* | *we* | ....

Noun phrases may be modified by adjectival phrases (ADJP; e.g., *the small Russian dog*) and numbers (Cd; e.g., *the five pastries*), each of which may optionally follow a determiner:

$$\text{NP} \rightarrow \text{ADJP Nn} \mid \text{ADJP Nns} \mid \text{Det ADJP Nn} \mid \text{Det ADJP Nns} \quad [9.34]$$

$$\text{NP} \rightarrow \text{Cd Nns} \mid \text{Det Cd Nns} \mid \ldots \quad [9.35]$$

Some noun phrases include multiple nouns, such as *the liberation movement* and *an antelope horn*, necessitating additional productions:

$$\text{NP} \rightarrow \text{Nn Nn} \mid \text{Nn Nns} \mid \text{Det Nn Nn} \mid \ldots \quad [9.36]$$

[8] Notice that the grammar does not include the recursive production S $\rightarrow$ ADVP S. It may be helpful to think about why this production would cause the grammar to overgenerate.

These multiple noun constructions can be combined with adjectival phrases and cardinal numbers, leading to a large number of additional productions.

Recursive noun phrase productions include coordination, prepositional phrase attachment, subordinate clauses, and verb phrase adjuncts:

NP → NP CC NP	*e.g., the red and the black*	[9.37]
NP → NP PP	*e.g., the President of the Georgia Institute of Technology*	[9.38]
NP → NP SBAR	*e.g., a whale which he had wounded*	[9.39]
NP → NP VP	*e.g., a whale taken near Shetland*	[9.40]

These recursive productions are a major source of ambiguity, because the VP and PP nonterminals can also generate NP children. Thus, the *the President of the Georgia Institute of Technology* can be derived in two ways, as can *a whale taken near Shetland in October*.

But aside from these few recursive productions, the noun phrase fragment of the Penn Treebank grammar is relatively flat, containing a large of number of productions that go from NP directly to a sequence of parts-of-speech. If noun phrases had more internal structure, the grammar would need fewer rules, which, as we will see, would make parsing faster and machine learning easier. Vadas and Curran (2011) propose to add additional structure in the form of a new non-terminal called a **nominal modifier** (NML), e.g.,

(9.6) a. (NP (NN crude) (NN oil) (NNS prices)) (PTB analysis)
b. (NP (NML (NN crude) (NN oil)) (NNS prices)) (NML-style analysis).

Another proposal is to treat the determiner as the head of a **determiner phrase** (DP; Abney, 1987). There are linguistic arguments for and against determiner phrases (e.g., Van Eynde, 2006). From the perspective of context-free grammar, DPs enable more structured analyses of some constituents, e.g.,

(9.7) a. (NP (DT the) (JJ white) (NN whale)) (PTB analysis)
b. (DP (DT the) (NP (JJ white) (NN whale))) (DP-style analysis).

Verb phrases

Verb phrases describe actions, events, and states of being. The PTB tagset distinguishes several classes of verb inflections: base form (VB; *she likes to snack*), present-tense third-person singular (VBZ; *she snacks*), present tense but not third-person singular (VBP; *they snack*), past tense (VBD; *they snacked*), present participle (VBG; *they are snacking*), and past participle (VBN; *they had snacked*).[9] Each of these forms can constitute a verb phrase on its

[9]This tagset is specific to English: for example, VBP is a meaningful category only because English morphology distinguishes third-person singular from all person-number combinations.

own:

$$\text{VP} \rightarrow \text{VB} \mid \text{VBZ} \mid \text{VBD} \mid \text{VBN} \mid \text{VBG} \mid \text{VBP} \quad [9.41]$$

More complex verb phrases can be formed by a number of recursive productions, including the use of coordination, modal verbs (MD; *she should snack*), and the infinitival *to* (TO):

VP → MD VP	*She* ***will snack***	[9.42]
VP → VBD VP	*She* ***had snacked***	[9.43]
VP → VBZ VP	*She* ***has been snacking***	[9.44]
VP → VBN VP	*She has* ***been snacking***	[9.45]
VP → TO VP	*She wants* ***to snack***	[9.46]
VP → VP CC VP	*She* ***buys and eats*** *many snacks*	[9.47]

Each of these productions uses recursion, with the VP non-terminal appearing in both the LHS and RHS. This enables the creation of complex verb phrases, such as *She will have wanted to have been snacking*.

Transitive verbs take noun phrases as direct objects, and ditransitive verbs take two direct objects:

VP → VBZ NP	*She* ***teaches algebra***	[9.48]
VP → VBG NP	*She has been* ***teaching algebra***	[9.49]
VP → VBD NP NP	*She* ***taught*** *her brother algebra*	[9.50]

These productions are *not* recursive, so a unique production is required for each verb part-of-speech. They also do not distinguish transitive from intransitive verbs, so the resulting grammar overgenerates examples like **She sleeps sushi* and **She learns Boyang algebra*. Sentences can also be direct objects:

VP → VBZ S	*Hunter* ***wants to eat the kimchi***	[9.51]
VP → VBZ SBAR	*Hunter* ***knows that Tristan ate the kimchi***	[9.52]

The first production overgenerates, licensing sentences like **Hunter sees Tristan eats the kimchi*. This problem could be addressed by designing a more specific set of sentence non-terminals, indicating whether the main verb can be conjugated.

Verbs can also be modified by prepositional phrases and adverbial phrases:

VP → VBZ PP	*She* ***studies at night***	[9.53]
VP → VBZ ADVP	*She* ***studies intensively***	[9.54]
VP → ADVP VBG	*She is* ***not studying***	[9.55]

Again, because these productions are not recursive, the grammar must include productions for every verb part-of-speech.

A special set of verbs, known as **copula**, can take **predicative adjectives** as direct objects:

VP → VBZ ADJP	*She* ***is hungry***	[9.56]
VP → VBP ADJP	*Success* ***seems increasingly unlikely***	[9.57]

The PTB does not have a special non-terminal for copular verbs, so this production generates non-grammatical examples such as **She eats tall*.

Particles (PRT as a phrase; RP as a part-of-speech) work to create phrasal verbs:

VP → VB PRT	*She told them to* ***fuck off***	[9.58]
VP → VBD PRT NP	*They* ***gave up their ill-gotten gains***	[9.59]

As the second production shows, particle productions are required for all configurations of verb parts-of-speech and direct objects.

Other contituents

The remaining constituents require far fewer productions. **Prepositional phrases** almost always consist of a preposition and a noun phrase,

PP → IN NP	*the whiteness* ***of the whale***	[9.60]
PP → TO NP	*What the white whale was* ***to Ahab****, has been hinted*	[9.61]

Similarly, complement clauses consist of a complementizer (usually a preposition, possibly null) and a sentence,

SBAR → IN S	*She said* ***that it was spicy***	[9.62]
SBAR → S	*She said* ***it was spicy***	[9.63]

Adverbial phrases are usually bare adverbs (ADVP → RB), with a few exceptions:

ADVP → RB RBR	*They went* ***considerably further***	[9.64]
ADVP → ADVP PP	*They went* ***considerably further than before***	[9.65]

The tag RBR is a comparative adverb.

Adjectival phrases extend beyond bare adjectives (ADJP → JJ) in a number of ways:

ADJP → RB JJ	*very hungry*	[9.66]
ADJP → RBR JJ	*more hungry*	[9.67]
ADJP → JJS JJ	*best possible*	[9.68]
ADJP → RB JJR	*even bigger*	[9.69]
ADJP → JJ CC JJ	*high and mighty*	[9.70]
ADJP → JJ JJ	*West German*	[9.71]
ADJP → RB VBN	*previously reported*	[9.72]

The tags JJR and JJS refer to comparative and superlative adjectives respectively.

All of these phrase types can be coordinated:

PP →PP CC PP	*on time and under budget*	[9.73]
ADVP →ADVP CC ADVP	*now and two years ago*	[9.74]
ADJP →ADJP CC ADJP	*quaint and rather deceptive*	[9.75]
SBAR →SBAR CC SBAR	*whether they want control or whether they want exports*	[9.76]

9.2.4 Grammatical ambiguity

Context-free parsing is useful not only because it determines whether a sentence is grammatical, but mainly because the constituents and their relations can be applied to tasks such as information extraction (chapter 17) and sentence compression (Jing, 2000; Clarke and Lapata, 2008). However, the **ambiguity** of wide-coverage natural language grammars poses a serious problem for such potential applications. As an example, Figure 9.13 shows two possible analyses for the simple sentence *We eat sushi with chopsticks*, depending on whether the *chopsticks* modify *eat* or *sushi*. Realistic grammars can license thousands or even millions of parses for individual sentences. **Weighted context-free grammars** solve this problem by attaching weights to each production, and selecting the derivation with the highest score. This is the focus of chapter 10.

9.3 *Mildly context-sensitive languages

Beyond context-free languages lie **context-sensitive languages**, in which the expansion of a non-terminal depends on its neighbors. In the general class of context-sensitive languages, computation becomes much more challenging: the membership problem for context-sensitive languages is PSPACE-complete. Since PSPACE contains the complexity class NP (problems that can be solved in polynomial time on a non-deterministic Turing

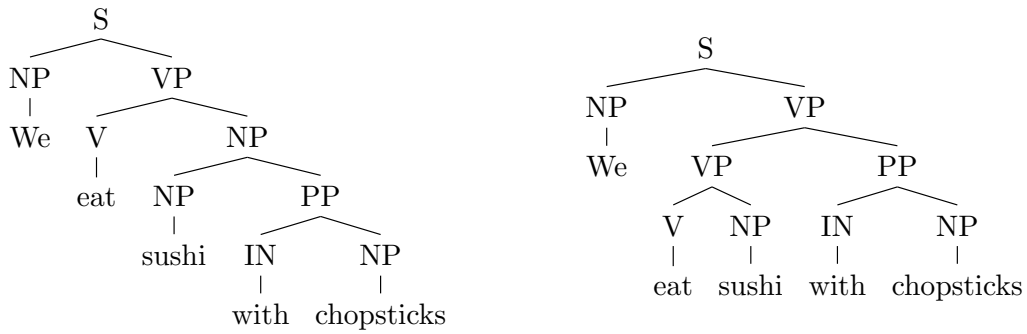


Figure 9.13: Two derivations of the same sentence

machine), PSPACE-complete problems cannot be solved efficiently if P $\neq$ NP. Thus, designing an efficient parsing algorithm for the full class of context-sensitive languages is probably hopeless.[10]

However, Joshi (1985) identifies a set of properties that define **mildly context-sensitive languages**, which are a strict subset of context-sensitive languages. Like context-free languages, mildly context-sensitive languages are parseable in polynomial time. However, the mildly context-sensitive languages include non-context-free languages, such as the "copy language" $\{\boldsymbol{ww} \mid \boldsymbol{w} \in \Sigma^*\}$ and the language $a^m b^n c^m d^n$. Both are characterized by **cross-serial dependencies**, linking symbols at long distance across the string.[11] For example, in the language $a^n b^m c^n d^m$, each a symbol is linked to exactly one c symbol, regardless of the number of intervening b symbols.

9.3.1 Context-sensitive phenomena in natural language

Such phenomena are occasionally relevant to natural language. A classic example is found in Swiss-German (Shieber, 1985), in which sentences such as *we let the children help Hans paint the house* are realized by listing all nouns before all verbs, i.e., *we the children Hans the house let help paint*. Furthermore, each noun's determiner is dictated by the noun's **case marking** (the role it plays with respect to the verb). Using an argument that is analogous to the earlier discussion of center-embedding (§ 9.2), Shieber describes these case marking constraints as a set of cross-serial dependencies, homomorphic to $a^m b^n c^m d^n$, and therefore not context-free.

[10] If PSPACE $\neq$ NP, then it contains problems that cannot be solved in polynomial time on a non-deterministic Turing machine; equivalently, solutions to these problems cannot even be checked in polynomial time (Arora and Barak, 2009).

[11] A further condition of the set of mildly-context-sensitive languages is *constant growth*: if the strings in the language are arranged by length, the gap in length between any pair of adjacent strings is bounded by some language specific constant. This condition excludes languages such as $\{a^{2^n} \mid n \geq 0\}$.

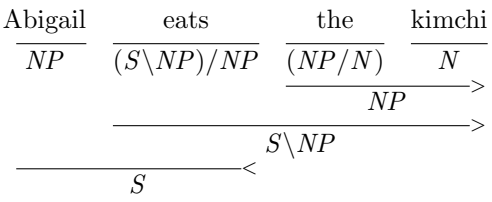


Figure 9.14: A syntactic analysis in CCG involving forward and backward function application

As with the move from regular to context-free languages, mildly context-sensitive languages can also be motivated by expedience. While finite sequences of cross-serial dependencies can in principle be handled in a context-free grammar, it is often more convenient to use a mildly context-sensitive formalism like **tree-adjoining grammar** (TAG) and **combinatory categorial grammar** (CCG). TAG-inspired parsers have been shown to be particularly effective in parsing the Penn Treebank (Collins, 1997; Carreras et al., 2008), and CCG plays a leading role in current research on semantic parsing (Zettlemoyer and Collins, 2005). These two formalisms are weakly equivalent: any language that can be specified in TAG can also be specified in CCG, and vice versa (Joshi et al., 1991). The remainder of the chapter gives a brief overview of CCG, but you are encouraged to consult Joshi and Schabes (1997) and Steedman and Baldridge (2011) for more detail on TAG and CCG respectively.

9.3.2 Combinatory categorial grammar

In combinatory categorial grammar, structural analyses are built up through a small set of generic combinatorial operations, which apply to immediately adjacent sub-structures. These operations act on the categories of the sub-structures, producing a new structure with a new category. The basic categories include S (sentence), NP (noun phrase), VP (verb phrase) and N (noun). The goal is to label the entire span of text as a sentence, S.

Complex categories, or types, are constructed from the basic categories, parentheses, and forward and backward slashes: for example, S/NP is a complex type, indicating a sentence that is lacking a noun phrase to its right; S\NP is a sentence lacking a noun phrase to its left. Complex types act as functions, and the most basic combinatory operations are function application to either the right or left neighbor. For example, the type of a verb phrase, such as *eats*, would be S\NP. Applying this function to a subject noun phrase to its left results in an analysis of *Abigail eats* as category S, indicating a successful parse.

Transitive verbs must first be applied to the direct object, which in English appears to the right of the verb, before the subject, which appears on the left. They therefore have the more complex type (S\NP)/NP. Similarly, the application of a determiner to the noun at

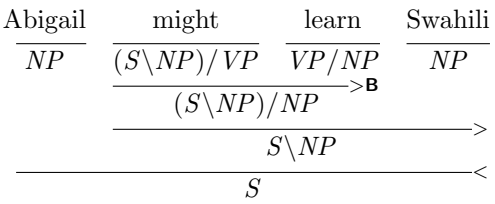


Figure 9.15: A syntactic analysis in CCG involving function composition (example modified from Steedman and Baldridge, 2011)

its right results in a noun phrase, so determiners have the type NP/N. Figure 9.14 provides an example involving a transitive verb and a determiner. A key point from this example is that it can be trivially transformed into phrase-structure tree, by treating each function application as a constituent phrase. Indeed, when CCG's only combinatory operators are forward and backward function application, it is equivalent to context-free grammar. However, the location of the "effort" has changed. Rather than designing good productions, the grammar designer must focus on the **lexicon** — choosing the right categories for each word. This makes it possible to parse a wide range of sentences using only a few generic combinatory operators.

Things become more interesting with the introduction of two additional operators: **composition** and **type-raising**. Function composition enables the combination of complex types: $X/Y \circ Y/Z \Rightarrow_{\mathbf{B}} X/Z$ (forward composition) and $Y\backslash Z \circ X\backslash Y \Rightarrow_{\mathbf{B}} X\backslash Z$ (backward composition).[12] Composition makes it possible to "look inside" complex types, and combine two adjacent units if the "input" for one is the "output" for the other. Figure 9.15 shows how function composition can be used to handle modal verbs. While this sentence can be parsed using only function application, the composition-based analysis is preferable because the unit *might learn* functions just like a transitive verb, as in the example *Abigail studies Swahili*. This in turn makes it possible to analyze conjunctions such as *Abigail studies and might learn Swahili*, attaching the direct object *Swahili* to the entire conjoined verb phrase *studies and might learn*. The Penn Treebank grammar fragment from § 9.2.3 would be unable to handle this case correctly: the direct object *Swahili* could attach only to the second verb *learn*.

Type raising converts an element of type X to a more complex type: $X \Rightarrow_T T/(T\backslash X)$ (forward type-raising to type T), and $X \Rightarrow_T T\backslash(T/X)$ (backward type-raising to type T). Type-raising makes it possible to reverse the relationship between a function and its argument — by transforming the argument into a function over functions over arguments! An example may help. Figure 9.15 shows how to analyze an object relative clause, *a story that Abigail tells*. The problem is that *tells* is a transitive verb, expecting a direct object to its right. As a result, *Abigail tells* is not a valid constituent. The issue is resolved by raising

[12]The subscript $\mathbf{B}$ follows notation from Curry and Feys (1958).

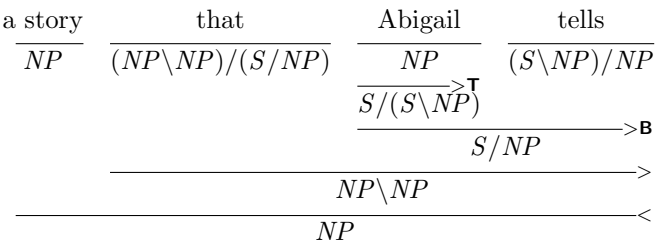


Figure 9.16: A syntactic analysis in CCG involving an object relative clause

Abigail from NP to the complex type (S/NP)\NP). This function can then be combined with the transitive verb *tells* by forward composition, resulting in the type (S/NP), which is a sentence lacking a direct object to its right.[13] From here, we need only design the lexical entry for the complementizer *that* to expect a right neighbor of type (S/NP), and the remainder of the derivation can proceed by function application.

Composition and type-raising give CCG considerable power and flexibility, but at a price. The simple sentence *Abigail tells Max* can be parsed in two different ways: by function application (first forming the verb phrase *tells Max*), and by type-raising and composition (first forming the non-constituent *Abigail tells*). This **derivational ambiguity** does not affect the resulting linguistic analysis, so it is sometimes known as **spurious ambiguity**. Hockenmaier and Steedman (2007) present a translation algorithm for converting the Penn Treebank into CCG derivations, using composition and type-raising only when necessary.

Exercises

1. Sketch out the state diagram for finite-state acceptors for the following languages on the alphabet $\{a, b\}$.

   a) Even-length strings. (Be sure to include 0 as an even number.)

   b) Strings that contain aaa as a substring.

   c) Strings containing an even number of a and an odd number of b symbols.

   d) Strings in which the substring bbb must be terminal if it appears — the string need not contain bbb, but if it does, nothing can come after it.

2. Levenshtein edit distance is the number of insertions, substitutions, or deletions required to convert one string to another.

[13] The missing direct object would be analyzed as a **trace** in CFG-like approaches to syntax, including the Penn Treebank.

a) Define a finite-state acceptor that accepts all strings with edit distance 1 from the target string, *target*.

b) Now think about how to generalize your design to accept all strings with edit distance from the target string equal to d. If the target string has length ℓ, what is the minimal number of states required?

3. Construct an FSA in the style of Figure 9.3, which handles the following examples:

   - *nation*/N, *national*/ADJ, *nationalize*/V, *nationalizer*/N
   - *America*/N, *American*/ADJ, *Americanize*/V, *Americanizer*/N

   Be sure that your FSA does not accept any further derivations, such as **nationalizeral* and **Americanizern*.

4. Show how to construct a trigram language model in a weighted finite-state acceptor. Make sure that you handle the edge cases at the beginning and end of the input.

5. Extend the FST in Figure 9.6 to handle the other two parts of rule 1a of the Porter stemmer: *-sses* $\rightarrow$ *ss*, and *-ies* $\rightarrow$ *-i*.

6. § 9.1.4 describes T_O, a transducer that captures English orthography by transducing *cook* + *ed* $\rightarrow$ *cooked* and *bake* + *ed* $\rightarrow$ *baked*. Design an unweighted finite-state transducer that captures this property of English orthography.

   Next, augment the transducer to appropriately model the suffix *-s* when applied to words ending in *s*, e.g. *kiss+s* $\rightarrow$ *kisses*.

7. Add parenthesization to the grammar in Figure 9.11 so that it is no longer ambiguous.

8. Construct three examples — a noun phrase, a verb phrase, and a sentence — which can be derived from the Penn Treebank grammar fragment in § 9.2.3, yet are not grammatical. Avoid reusing examples from the text. Optionally, propose corrections to the grammar to avoid generating these cases.

9. Produce parses for the following sentences, using the Penn Treebank grammar fragment from § 9.2.3.

   (9.8) This aggression will not stand.

   (9.9) I can get you a toe.

   (9.10) Sometimes you eat the bar and sometimes the bar eats you.

   Then produce parses for three short sentences from a news article from this week.

10. * One advantage of CCG is its flexibility in handling coordination:

    (9.11) a. *Hunter and Tristan speak Hawaiian*
    b. *Hunter speaks and Tristan understands Hawaiian*

    Define the lexical entry for *and* as

    $$\textit{and} := (X/X)\backslash X, \tag{9.77}$$

    where X can refer to any type. Using this lexical entry, show how to parse the two examples above. In the second example, *Swahili* should be combined with the coordination *Abigail speaks and Max understands*, and not just with the verb *understands*.