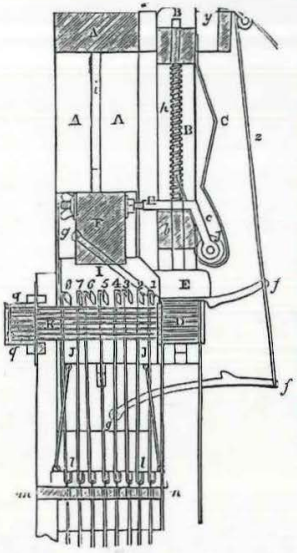
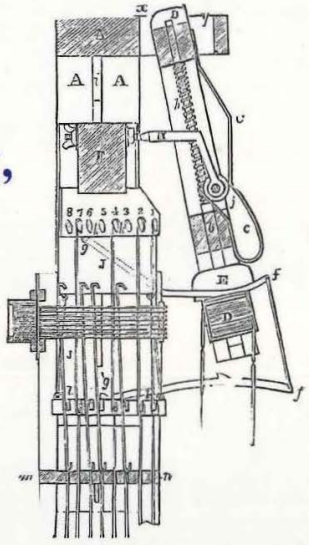


SPEECH and LANGUAGE PROCESSING

An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition



DANIEL JURAFSKY & JAMES H. MARTIN

Speech and Language Processing

"This book is an absolute necessity for instructors at all levels, as well as an indispensible reference for researchers. Introducing NLP, computational linguistics, and speech recognition comprehensively in a single book is an ambitious enterprise. The authors have managed it admirably, paying careful attention to traditional foundations, relating recent developments and trends to those foundations, and tying it all together with insight and humor. Remarkable."

– Philip Resnik, University of Maryland

"...ideal for ... linguists who want to learn more about computational modeling and techniques in language processing; computer scientists building language applications who want to learn more about the linguistic underpinnings of the field; speech technologists who want to learn more about language understanding, semantics and discourse; and all those wanting to learn more about speech processing. For instructors ... this book is a dream. It covers virtually every aspect of NLP... What's truly astounding is that the book covers such a broad range of topics, while giving the reader the depth to understand and make use of the concepts, algorithms and techniques that are presented... ideal as a course textbook for advanced undergraduates, as well as graduate students and researchers in the field.

– Johanna Moore, University of Edinburgh

"*Speech and Language Processing* is a comprehensive, reader-friendly, and up-to-date guide to computational linguistics, covering both statistical and symbolic methods and their application. It will appeal both to senior undergraduate students, who will find it neither too technical nor too simplistic, and to researchers, who will find it to be a helpful guide to the newly established techniques of a rapidly growing research field."

– Graeme Hirst, University of Toronto

"The field of human language processing encompasses a diverse array of disciplines, and as such is an incredibly challenging field to master. This book does a wonderful job of bringing together this vast body of knowledge in a form that is both accessible and comprehensive. Its encyclopedic coverage makes it a must-have for people already in the field, while the clear presentation style and many examples make it an ideal textbook."

– Eric Brill, Microsoft Research

This is quite simply the most complete introduction to natural language and speech technology ever written. Virtually every topic in the field is covered, in a prose style that is both clear and engaging. The discussion is linguistically informed, and strikes a nice balance between theoretical computational models, and practical applications. It is an extremely impressive achievement.

– Richard Sproat, AT&T Labs – Research

PRENTICE HALL SERIES
IN ARTIFICIAL INTELLIGENCE
Stuart Russell and Peter Norvig, Editors

GRAHAM	ANSI *Common Lisp*
RUSSELL & NORVIG	*Artificial Intelligence: A Modern Approach*
JURAFSKY & MARTIN	*Speech and Language Processing*

Speech and Language Processing

An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition

Daniel Jurafsky and James H. Martin

University of Colorado, Boulder

Contributing writers:
Andrew Kehler, Keith Vander Linden, and Nigel Ward

Prentice Hall
Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Jurafsky, Daniel S. (Daniel Saul)
Speech and Langauge Processing / Daniel Jurafsky, James H. Martin.
p. cm.

Includes bibliographical references and index.
ISBN 0-13-095069-6

Editor-in-Chief: *Marcia Horton*
Publisher: *Alan Apt*
Editorial/production supervision: *Scott Disanno*
Editorial assistant: *Toni Holm*
Executive managing editor: *Vince O'Brien*
Cover design director: *Heather Scott*
Cover design execution: *John Christiana*
Manufacturing manager: *Trudy Pisciotti*
Manufacturing buyer: *Pat Brown*
Assistant vice-president of production and manufacturing: *David W. Riccardi*

Cover design: *Daniel Jurafsky, James H. Martin, and Linda Martin.* The front cover drawing is the action for the Jacquard Loom (Usher, 1954). The back cover drawing is Alexander Graham Bell's Gallows telephone (Rhodes, 1929).

This book was set in Times-Roman, TIPA (IPA), and PMC (Chinese) by the authors using LaTeX2e.

Pearson Higher Education
Upper Saddle River, New Jersey 07458

Printed in the United States of America

10 9 8 7 6 5 4 3
ISBN 0-13-095069-6

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

For my parents, Ruth and Al Jurafsky — D.J.

For Linda — J.M.

Summary of Contents

Contents

Foreword

Linguistics has a hundred-year history as a scientific discipline, and computational linguistics has a forty-year history as a part of computer science. But it is only in the last five years that language understanding has emerged as an industry reaching millions of people, with information retrieval and machine translation available on the internet, and speech recognition becoming popular on desktop computers. This industry has been enabled by theoretical advances in the representation and processing of language information.

Speech and Language Processing is the first book to thoroughly cover language technology, at all levels and with all modern technologies. It combines deep linguistic analysis with robust statistical methods. From the point of view of levels, the book starts with the word and its components, moving up to the way words fit together (or syntax), to the meaning (or semantics) of words, phrases and sentences, and concluding with issues of coherent texts, dialog, and translation. From the point of view of technologies, the book covers regular expressions, information retrieval, context free grammars, unification, first-order predicate calculus, hidden Markov and other probabilistic models, rhetorical structure theory, and others. Previously you would need two or three books to get this kind of coverage. *Speech and Language Processing* covers the full range in one book, but more importantly, it relates the technologies to each other, giving the reader a sense of how each one is best used, and how they can be used together. It does all this with an engaging style that keeps the reader's interest and motivates the technical details in a way that is thorough but not dry. Whether you're interested in the field from the scientific or the industrial point of view, this book serves as an ideal introduction, reference, and guide to future study of this fascinating field.

Peter Norvig & Stuart Russell, Editors
Prentice Hall Series in Artificial Intelligence

Preface

This is an exciting time to be working in speech and language processing. Historically distinct fields (natural language processing, speech recognition, computational linguistics, computational psycholinguistics) have begun to merge. The commercial availability of speech recognition and the need for Web-based language techniques have provided an important impetus for development of real systems. The availability of very large on-line corpora has enabled statistical models of language at every level, from phonetics to discourse. We have tried to draw on this emerging state of the art in the design of this pedagogical and reference work:

1. *Coverage*
   In attempting to describe a unified vision of speech and language processing, we cover areas that traditionally are taught in different courses in different departments: speech recognition in electrical engineering; parsing, semantic interpretation, and pragmatics in natural language processing courses in computer science departments; and computational morphology and phonology in computational linguistics courses in linguistics departments. The book introduces the fundamental algorithms of each of these fields, whether originally proposed for spoken or written language, whether logical or statistical in origin, and attempts to tie together the descriptions of algorithms from different domains. We have also included coverage of applications like spelling-checking and information retrieval and extraction as well as areas like cognitive modeling. A potential problem with this broad-coverage approach is that it required us to include introductory material for each field; thus linguists may want to skip our description of articulatory phonetics, computer scientists may want to skip such sections as regular expressions, and electrical engineers skip the sections on signal processing. Of course, even in a book this long, we didn't have room for everything. Thus this book should not be considered a substitute for important relevant courses in linguistics, automata and formal language theory, or, especially, statistics and information theory.
2. *Emphasis on practical applications*
   It is important to show how language-related algorithms and techniques (from HMMs to unification, from the lambda calculus to transformation-based learning) can be applied to important real-world problems: spelling checking, text document search, speech recogni-

tion, Web-page processing, part-of-speech tagging, machine translation, and spoken-language dialogue agents. We have attempted to do this by integrating the description of language processing applications into each chapter. The advantage of this approach is that as the relevant linguistic knowledge is introduced, the student has the background to understand and model a particular domain.

3. *Emphasis on scientific evaluation*
   The recent prevalence of statistical algorithms in language processing and the growth of organized evaluations of speech and language processing systems has led to a new emphasis on evaluation. We have, therefore, tried to accompany most of our problem domains with a **Methodology Box** describing how systems are evaluated (e.g., including such concepts as training and test sets, cross-validation, and information-theoretic evaluation metrics like perplexity).
4. *Description of widely available language processing resources*
   Modern speech and language processing is heavily based on common resources: raw speech and text corpora, annotated corpora and treebanks, standard tagsets for labeling pronunciation, part-of-speech, parses, word-sense, and dialogue-level phenomena. We have tried to introduce many of these important resources throughout the book (e.g., the Brown, Switchboard, callhome, ATIS, TREC, MUC, and BNC corpora) and provide complete listings of many useful tagsets and coding schemes (such as the Penn Treebank, CLAWS C5 and C7, and the ARPAbet) but some inevitably got left out. Furthermore, rather than include references to URLs for many resources directly in the textbook, we have placed them on the book's Web site, where they can more readily updated.

The book is primarily intended for use in a graduate or advanced undergraduate course or sequence. Because of its comprehensive coverage and the large number of algorithms, the book is also useful as a reference for students and professionals in any of the areas of speech and language processing.

Overview of the Book

The book is divided into four parts in addition to an introduction and end matter. Part I, "Words", introduces concepts related to the processing of words: phonetics, phonology, morphology, and algorithms used to process them: finite automata, finite transducers, weighted transducers, N-grams,

and Hidden Markov Models. Part II, "Syntax", introduces parts-of-speech and phrase structure grammars for English and gives essential algorithms for processing word classes and structured relationships among words: part-of-speech taggers based on HMMs and transformation-based learning, the CYK and Earley algorithms for parsing, unification and typed feature structures, lexicalized and probabilistic parsing, and analytical tools like the Chomsky hierarchy and the pumping lemma. Part III, "Semantics", introduces first order predicate calculus and other ways of representing meaning, several approaches to compositional semantic analysis, along with applications to information retrieval, information extraction, speech understanding, and machine translation. Part IV, "Pragmatics", covers reference resolution and discourse structure and coherence, spoken dialogue phenomena like dialogue and speech act modeling, dialogue structure and coherence, and dialogue managers, as well as a comprehensive treatment of natural language generation and of machine translation.

Using this Book

The book provides enough material to be used for a full-year sequence in speech and language processing. It is also designed so that it can be used for a number of different useful one-term courses:

NLP 1 quarter	NLP 1 semester	Speech + NLP 1 semester	Comp. Linguistics 1 quarter
1. Intro	1. Intro	1. Intro	1. Intro
2. Regex, FSA	2. Regex, FSA	2. Regex, FSA	2. Regex, FSA
8. POS tagging	3. Morph., FST	3. Morph., FST	3. Morph., FST
9. CFGs	6. *N*-grams	4. Comp. Phonol.	4. Comp. Phonol.
10. Parsing	8. POS tagging	5. Prob. Pronun.	10. Parsing
11. Unification	9. CFGs	6. *N*-grams	11. Unification
14. Semantics	10. Parsing	7. HMMs & ASR	13. Complexity
15. Sem. Analysis	11. Unification	8. POS tagging	16. Lex. Semantics
18. Discourse	12. Prob. Parsing	9. CFGs	18. Discourse
20. Generation	14. Semantics	10. Parsing	19. Dialogue
	15. Sem. Analysis	12. Prob. Parsing	
	16. Lex. Semantics	14. Semantics	
	17. WSD and IR	15. Sem. Analysis	
	18. Discourse	19. Dialogue	
	20. Generation	21. Mach. Transl.	
	21. Mach. Transl.		

Selected chapters from the book could also be used to augment courses in Artificial Intelligence, Cognitive Science, or Information Retrieval.

Acknowledgments

The three contributing writers for the book are Andy Kehler, who wrote Chapter 18 (Discourse), Keith Vander Linden, who wrote Chapter 20 (Generation), and Nigel Ward, who wrote most of Chapter 21 (Machine Translation). Andy Kehler also wrote Section 19.4 of Chapter 19. Paul Taylor wrote most of Section 4.7 and Section 7.8.

Dan would like to thank his parents for encouraging him to do a really good job of everything he does, finish it in a timely fashion, and make time for going to the gym. He would also like to thank Nelson Morgan, for introducing him to speech recognition and teaching him to ask "but does it work?"; Jerry Feldman, for sharing his intense commitment to finding the right answers and teaching him to ask "but is it really important?"; Chuck Fillmore, his first advisor, for sharing his love for language and especially argument structure, and teaching him to always go look at the data, (and all of them for teaching by example that it's only worthwhile if it's fun); and Robert Wilensky, his dissertation advisor, for teaching him the importance of collaboration and group spirit in research. He is also grateful to the CU Lyric Theater program and the casts of *South Pacific*, *Gianni Schicchi*, *Guys and Dolls*, *Gondoliers*, *Iolanthe*, and *Oklahoma*, and to Doris and Cary, Elaine and Eric, Irene and Sam, Susan and Richard, Lisa and Mike, Mike and Fia, Erin and Chris, Eric and Beth, Pearl and Tristan, Bruce and Peggy, Ramon and Rebecca, Adele and Ali, Terry, Kevin, Becky, Temmy, Lil, Lin and Ron and David, Mike, and Jessica and Bill, and all their families for providing lots of emotional support and often a place to stay during the writing.

Jim would like to thank his parents for encouraging him and allowing him to follow what must have seemed like an odd path at the time. He would also like to thank his thesis advisor, Robert Wilensky, for giving him his start in NLP at Berkeley; Peter Norvig, for providing many positive examples along the way; Rick Alterman, for encouragement and inspiration at a critical time; and Chuck Fillmore, George Lakoff, Paul Kay, and Susanna Cumming for teaching him what little he knows about linguistics. He'd also like to thank Michael Main for covering for him while he shirked his departmental duties. Finally, he'd like to thank his wife Linda for all her support and patience through all the years it took to complete this book.

Boulder is a very rewarding place to work on speech and language processing. We'd like to thank our colleagues here for their collaborations, which have greatly influenced our research and teaching: Alan Bell, Barbara Fox, Laura Michaelis and Lise Menn in linguistics; Clayton Lewis, Gerhard

Fischer, Mike Eisenberg, Mike Mozer, Liz Jessup, and Andrzej Ehrenfeucht in computer science; Walter Kintsch, Tom Landauer, and Alice Healy in psychology; Ron Cole, John Hansen, and Wayne Ward in the Center for Spoken Language Understanding, and our current and former students in the computer science and linguistics departments: Marion Bond, Noah Coccaro, Michelle Gregory, Keith Herold, Michael Jones, Patrick Juola, Keith Vander Linden, Laura Mather, Taimi Metzler, Douglas Roland, and Patrick Schone.

This book has benefited from careful reading and enormously helpful comments from a number of readers and from course-testing. We are deeply indebted to colleagues who each took the time to read and give extensive comments and advice, which vastly improved large parts of the book, including Alan Bell, Bob Carpenter, Jan Daciuk, Graeme Hirst, Andy Kehler, Kemal Oflazer, Andreas Stolcke, and Nigel Ward. Our editor Alan Apt, our series editors Peter Norvig and Stuart Russell, and our production editor Scott DiSanno made many helpful suggestions on design and content. We are also indebted to many friends and colleagues who read individual sections of the book or answered our many questions for their comments and advice, including the students in our classes at the University of Colorado, Boulder, and in Dan's classes at the University of California, Berkeley, and the LSA Summer Institute at the University of Illinois at Urbana-Champaign, as well as

Yoshi Asano, Todd M. Bailey, John Bateman, Giulia Bencini, Lois Boggess, Michael Braverman, Nancy Chang, Jennifer Chu-Carroll, Noah Coccaro, Gary Cottrell, Gary Dell, Jeff Elman, Robert Dale, Dan Fass, Bill Fisher, Eric Fosler-Lussier, James Garnett, Susan Garnsey, Dale Gerdemann, Dan Gildea, Michelle Gregory, Nizar Habash, Jeffrey Haemer, Jorge Hankamer, Keith Herold, Beth Heywood, Derrick Higgins, Erhard Hinrichs, Julia Hirschberg, Jerry Hobbs, Fred Jelinek, Liz Jessup, Aravind Joshi, Terry Kleeman, Jean-Pierre Koenig, Kevin Knight, Shalom Lappin, Julie Larson, Stephen Levinson, Jim Magnuson, Jim Mayfield, Lise Menn, Laura Michaelis, Corey Miller, Nelson Morgan, Christine Nakatani, Mike Neufeld, Peter Norvig, Mike O'Connell, Mick O'Donnell, Rob Oberbreckling, Martha Palmer, Dragomir Radev, Terry Regier, Ehud Reiter, Phil Resnik, Klaus Ries, Ellen Riloff, Mike Rosner, Dan Roth, Patrick Schone, Liz Shriberg, Richard Sproat, Subhashini Srinivasin, Paul Taylor, Wayne Ward, Pauline Welby, Dekai Wu, and Victor Zue.

We'd also like to thank the Institute of Cognitive Science and the Departments of Computer Science and Linguistics for their support over the years. We are also very grateful to the National Science Foundation: Dan Jurafsky's time on the book was supported in part by NSF CAREER Award IIS-9733067 and Andy Kehler was supported in part by NSF Award IIS-9619126.

Daniel Jurafsky
James H. Martin
Boulder, Colorado

1 INTRODUCTION

Dave Bowman: Open the pod bay doors, HAL.
HAL: I'm sorry Dave, I'm afraid I can't do that.
Stanley Kubrick and Arthur C. Clarke,
screenplay of *2001: A Space Odyssey*

The HAL 9000 computer in Stanley Kubrick's film *2001: A Space Odyssey* is one of the most recognizable characters in twentieth-century cinema. HAL is an artificial agent capable of such advanced language-processing behavior as speaking and understanding English, and at a crucial moment in the plot, even reading lips. It is now clear that HAL's creator Arthur C. Clarke was a little optimistic in predicting when an artificial agent such as HAL would be available. But just how far off was he? What would it take to create at least the language-related parts of HAL? Minimally, such an agent would have to be capable of interacting with humans via language, which includes understanding humans via **speech recognition** and **natural language understanding** (and, of course, **lip-reading**), and of communicating with humans via **natural language generation** and **speech synthesis**. HAL would also need to be able to do **information retrieval** (finding out where needed textual resources reside), **information extraction** (extracting pertinent facts from those textual resources), and **inference** (drawing conclusions based on known facts).

Although these problems are far from completely solved, much of the language-related technology that HAL needs is currently being developed, with some of it already available commercially. Solving these problems, and others like them, is the main concern of the fields known as Natural Language Processing, Computational Linguistics, and Speech Recognition and Synthesis, which together we call **Speech and Language Processing**. The goal of this book is to describe the state of the art of this technology

at the start of the twenty-first century. The applications we will consider are all of those needed for agents like HAL as well as other valuable areas of language processing such as **spelling correction**, **grammar checking**, **information retrieval**, and **machine translation**.

1.1 KNOWLEDGE IN SPEECH AND LANGUAGE PROCESSING

By speech and language processing, we have in mind those computational techniques that process spoken and written human language, *as language*. As we will see, this is an inclusive definition that encompasses everything from mundane applications such as word counting and automatic hyphenation, to cutting edge applications such as automated question answering on the Web, and real-time spoken language translation.

What distinguishes these language processing applications from other data processing systems is their use of *knowledge of language*. Consider the Unix `wc` program, which is used to count the total number of bytes, words, and lines in a text file. When used to count bytes and lines, `wc` is an ordinary data processing application. However, when it is used to count the words in a file it requires *knowledge about what it means to be a word*, and thus becomes a language processing system.

Of course, `wc` is an extremely simple system with an extremely limited and impoverished knowledge of language. More-sophisticated language agents such as HAL require much broader and deeper knowledge of language. To get a feeling for the scope and kind of knowledge required in more-sophisticated applications, consider some of what HAL would need to know to engage in the dialogue that begins this chapter.

To determine what Dave is saying, HAL must be capable of analyzing an incoming audio signal and recovering the exact sequence of words Dave used to produce that signal. Similarly, in generating its response, HAL must be able to take a sequence of words and generate an audio signal that Dave can recognize. Both of these tasks require knowledge about **phonetics and phonology**, which can help model how words are pronounced in colloquial speech (Chapters 4 and 5).

Note also that unlike Star Trek's Commander Data, HAL is capable of producing contractions like *I'm* and *can't*. Producing and recognizing these and other variations of individual words (e.g., recognizing that *doors* is plural) requires knowledge about **morphology**, which captures information about the shape and behavior of words in context (Chapters 2 and 3).

Moving beyond individual words, HAL must know how to analyze the structure underlying Dave's request. Such an analysis is necessary among other reasons for HAL to determine that Dave's utterance is a request for action, as opposed to a simple statement about the world or a question about the door, as in the following variations of his original statement.

HAL, the pod bay door is open.

HAL, is the pod bay door open?

In addition, HAL must use similar structural knowledge to properly string together the words that constitute its response. For example, HAL must know that the following sequence of words will not make sense to Dave, despite the fact that it contains precisely the same set of words as the original.

I'm I do, sorry that afraid Dave I'm can't.

The knowledge needed to order and group words together comes under the heading of **syntax**.

Of course, simply knowing the words and the syntactic structure of what Dave said does not tell HAL much about the nature of his request. To know that Dave's command is actually about opening the pod bay door, rather than an inquiry about the day's lunch menu, requires knowledge of the meanings of the component words, the domain of **lexical semantics**, and knowledge of how these components combine to form larger meanings, **compositional semantics**.

Next, despite its bad behavior, HAL knows enough to be polite to Dave. It could, for example, have simply replied *No* or *No, I won't open the door*. Instead, it first embellishes its response with the phrases *I'm sorry* and *I'm afraid*, and then only indirectly signals its refusal by saying *I can't*, rather than the more direct (and truthful) *I won't*.[1] The appropriate use of this kind of polite and indirect language comes under the heading of **pragmatics**.

Finally, rather than simply ignoring Dave's command and leaving the door closed, HAL chooses to engage in a structured conversation relevant to Dave's initial request. HAL's correct use of the word *that* in its answer to Dave's request is a simple illustration of the kind of between-utterance device common in such conversations. Correctly structuring these such conversations requires knowledge of **discourse conventions**.

To summarize, the knowledge of language needed to engage in complex language behavior can be separated into six distinct categories.

[1] For those unfamiliar with HAL, it is neither sorry nor afraid, nor is it incapable of opening the door. It has simply decided in a fit of paranoia to kill its crew.

- Phonetics and Phonology — The study of linguistic sounds
- Morphology — The study of the meaningful components of words
- Syntax — The study of the structural relationships between words
- Semantics — The study of meaning
- Pragmatics — The study of how language is used to accomplish goals
- Discourse — The study of linguistic units larger than a single utterance

1.2 AMBIGUITY

AMBIGUITY

A perhaps surprising fact about the six categories of linguistic knowledge is that most or all tasks in speech and language processing can be viewed as resolving **ambiguity** at one of these levels. We say some input is ambiguous if there are multiple alternative linguistic structures than can be built for it. Consider the spoken sentence *I made her duck.* Here's five different meanings this sentence could have (there are more), each of which exemplifies an ambiguity at some level:

(1.1) I cooked waterfowl for her.

(1.2) I cooked waterfowl belonging to her.

(1.3) I created the (plaster?) duck she owns.

(1.4) I caused her to quickly lower her head or body.

(1.5) I waved my magic wand and turned her into undifferentiated waterfowl.

These different meanings are caused by a number of ambiguities. First, the words *duck* and *her* are morphologically or syntactically ambiguous in their part-of-speech. *Duck* can be a verb or a noun, while *her* can be a dative pronoun or a possessive pronoun. Second, the word *make* is semantically ambiguous; it can mean *create* or *cook*. Finally, the verb *make* is syntactically ambiguous in a different way. *Make* can be transitive, that is, taking a single direct object (1.2), or it can be ditransitive, that is, taking two objects (1.5), meaning that the first object (*her*) got made into the second object (*duck*). Finally, *make* can take a direct object and a verb (1.4), meaning that the object (*her*) got caused to perform the verbal action (*duck*). Furthermore, in a spoken sentence, there is an even deeper kind of ambiguity; the first word could have been *eye* or the second word *maid.*

We will often introduce the models and algorithms we present throughout the book as ways to **resolve** or **disambiguate** these ambiguities. For

example deciding whether *duck* is a verb or a noun can be solved by **part-of-speech tagging**. Deciding whether *make* means "create" or "cook" can be solved by **word sense disambiguation**. Resolution of part-of-speech and word sense ambiguities are two important kinds of **lexical disambiguation**. A wide variety of tasks can be framed as lexical disambiguation problems. For example, a text-to-speech synthesis system reading the word *lead* needs to decide whether it should be pronounced as in *lead pipe* or as in *lead me on*. By contrast, deciding whether *her* and *duck* are part of the same entity (as in (1.1) or (1.4)) or are different entity (as in (1.2)) is an example of **syntactic disambiguation** and can be addressed by **probabilistic parsing**. Ambiguities that don't arise in this particular example (like whether a given sentence is a statement or a question) will also be resolved, for example by **speech act interpretation**.

1.3 Models and Algorithms

One of the key insights of the last 50 years of research in language processing is that the various kinds of knowledge described in the last sections can be captured through the use of a small number of formal models, or theories. Fortunately, these models and theories are all drawn from the standard toolkits of Computer Science, Mathematics, and Linguistics and should be generally familiar to those trained in those fields. Among the most important elements in this toolkit are **state machines**, **formal rule systems**, **logic**, as well as **probability theory** and other machine learning tools. These models, in turn, lend themselves to a small number of algorithms from well-known computational paradigms. Among the most important of these are **state space search** algorithms and **dynamic programming** algorithms.

In their simplest formulation, state machines are formal models that consist of states, transitions among states, and an input representation. Some of the variations of this basic model that we will consider are **deterministic** and **non-deterministic finite-state automata**, **finite-state transducers**, which can write to an output device, **weighted automata**, **Markov models**, and **hidden Markov models**, which have a probabilistic component.

Closely related to these somewhat procedural models are their declarative counterparts: formal rule systems. Among the more important ones we will consider are **regular grammars** and **regular relations**, **context-free grammars**, **feature-augmented grammars**, as well as probabilistic variants of them all. State machines and formal rule systems are the main tools

used when dealing with knowledge of phonology, morphology, and syntax.

The algorithms associated with both state-machines and formal rule systems typically involve a search through a space of states representing hypotheses about an input. Representative tasks include searching through a space of phonological sequences for a likely input word in speech recognition, or searching through a space of trees for the correct syntactic parse of an input sentence. Among the algorithms that are often used for these tasks are well-known graph algorithms such as **depth-first search**, as well as heuristic variants such as **best-first**, and **A* search**. The dynamic programming paradigm is critical to the computational tractability of many of these approaches by ensuring that redundant computations are avoided.

The third model that plays a critical role in capturing knowledge of language is logic. We will discuss **first order logic**, also known as the **predicate calculus**, as well as such related formalisms as feature-structures, semantic networks, and conceptual dependency. These logical representations have traditionally been the tool of choice when dealing with knowledge of semantics, pragmatics, and discourse (although, as we will see, applications in these areas are increasingly relying on the simpler mechanisms used in phonology, morphology, and syntax).

Probability theory is the final element in our set of techniques for capturing linguistic knowledge. Each of the other models (state machines, formal rule systems, and logic) can be augmented with probabilities. One major use of probability theory is to solve the many kinds of ambiguity problems that we discussed earlier; almost any speech and language processing problem can be recast as: "given N choices for some ambiguous input, choose the most probable one".

Another major advantage of probabilistic models is that they are one of a class of **machine learning** models. Machine learning research has focused on ways to automatically learn the various representations described above; automata, rule systems, search heuristics, classifiers. These systems can be trained on large corpora and can be used as a powerful modeling technique, especially in places where we don't yet have good causal models. Machine learning algorithms will be described throughout the book.

1.4 Language, Thought, and Understanding

To many, the ability of computers to process language as skillfully as we do will signal the arrival of truly intelligent machines. The basis of this belief is

the fact that the effective use of language is intertwined with our general cognitive abilities. Among the first to consider the computational implications of this intimate connection was Alan Turing (1950). In this famous paper, Turing introduced what has come to be known as the **Turing Test**. Turing began with the thesis that the question of what it would mean for a machine to think was essentially unanswerable due to the inherent imprecision in the terms *machine* and *think*. Instead, he suggested an empirical test, a game, in which a computer's use of language would form the basis for determining if it could think. If the machine could win the game it would be judged intelligent.

TURING TEST

In Turing's game, there are three participants: two people and a computer. One of the people is a contestant and plays the role of an interrogator. To win, the interrogator must determine which of the other two participants is the machine by asking a series of questions via a teletype. The task of the machine is to fool the interrogator into believing it is a person by responding as a person would to the interrogator's questions. The task of the second human participant is to convince the interrogator that the other participant is the machine, and that they are human.

The following interaction from Turing's paper illustrates the kind of interactions he had in mind. Clearly a convincing simulation of humanity doesn't require being an expert in all domains:

Q: Please write me a sonnet on the topic of the Forth Bridge.
A: Count me out on this one. I never could write poetry.
Q: Add 34957 to 70764.
A: (Pause about 30 seconds and then give answer as) 105621.

Turing predicted that roughly by the end of the twentieth century a machine with 10 gigabytes of memory would have around a 30% chance of fooling a human interrogator after 5 minutes of questioning. Given the fact that you can fool some of the people all the time, it is not clear how rigorous this particular standard is. Nevertheless, the critical issue for Turing was that using language as humans do is sufficient, by itself, as an operational test for intelligence.

Surprisingly, many of the issues raised by Turing became relevant as early as 1966 in the context of the ELIZA program (Weizenbaum, 1966). ELIZA was an early natural language processing system capable of carrying on a limited form of conversation with a user. Consider the following session with a version of ELIZA that imitated the responses of a Rogerian psychotherapist.

$User_1$: You are like my father in some ways.
$ELIZA_1$: WHAT RESEMBLANCE DO YOU SEE
$User_2$: You are not very aggressive but I think you don't want me to notice that.
$ELIZA_2$: WHAT MAKES YOU THINK I AM NOT AGGRESSIVE
$User_3$: You don't argue with me.
$ELIZA_3$: WHY DO YOU THINK I DON'T ARGUE WITH YOU
$User_4$: You are afraid of me.
$ELIZA_4$: DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU

As we will discuss in more detail in Chapter 2, ELIZA is a remarkably simple program that makes use of pattern-matching to process the input and translate it into suitable outputs. The success of this simple technique in this domain is due to the fact that ELIZA doesn't actually need to *know* anything to mimic a Rogerian psychotherapist. As Weizenbaum notes, this is one of the few dialogue genres where the listener can act as if they know nothing of the world.

ELIZA's deep relevance to Turing's ideas is that many people who interacted with ELIZA came to believe that it really *understood* them and their problems. Indeed, Weizenbaum (1976) notes that many of these people continued to believe in ELIZA's abilities even after the program's operation was explained to them. In more recent years, Weizenbaum's informal reports have been repeated in a somewhat more controlled setting. Since 1991, an event known as the Loebner Prize competition has attempted to put various computer programs to the Turing test. Although these contests have proven to have little scientific interest, a consistent result over the years has been that even the crudest programs can fool some of the judges some of the time (Shieber, 1994). Not surprisingly, these results have done nothing to quell the ongoing debate over the suitability of the Turing test as a test for intelligence among philosophers and AI researchers (Searle, 1980).

Fortunately, for the purposes of this book, the relevance of these results does not hinge on whether or not computers will ever be intelligent, or understand natural language. Far more important is recent related research in the social sciences that has confirmed another of Turing's predictions from the same paper.

> Nevertheless I believe that at the end of the century the use of words and educated opinion will have altered so much that we will be able to speak of machines thinking without expecting to be contradicted.

It is now clear that regardless of what people believe or know about the inner workings of computers, they talk about them and interact with them as

social entities. People act toward computers as if they were people; they are polite to them, treat them as team members, and expect among other things that computers should be able to understand their needs, and be capable of interacting with them naturally. For example, Reeves and Nass (1996) found that when a computer asked a human to evaluate how well the computer had been doing, the human gives more positive responses than when a different computer asks the same questions. People seemed to be afraid of being impolite. In a different experiment, Reeves and Nass found that people also give computers higher performance ratings if the computer has recently said something flattering to the human. Given these predispositions, speech and language-based systems may provide many users with the most natural interface for many applications. This fact has led to a long-term focus in the field on the design of **conversational agents**, artificial entities that communicate conversationally.

1.5 The State of the Art and the Near-Term Future

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> Alan Turing.

This is an exciting time for the field of speech and language processing. The recent commercialization of robust speech recognition systems, and the rise of the Web, have placed speech and language processing applications in the spotlight, and have pointed out a plethora of exciting possible applications. The following scenarios serve to illustrate some current applications and near-term possibilities.

A Canadian computer program accepts daily weather data and generates weather reports that are passed along unedited to the public in English and French (Chandioux, 1976).

The *Babel Fish* translation system from Systran handles over 1,000,000 translation requests a day from the AltaVista search engine site.

A visitor to Cambridge, Massachusetts, asks a computer about places to eat using only spoken language. The system returns relevant information from a database of facts about the local restaurant scene (Zue et al., 1991).

These scenarios represent just a few of applications possible given current technology. The following, somewhat more speculative scenarios, give

some feeling for applications currently being explored at research and development labs around the world.

A computer reads hundreds of typed student essays and grades them in a manner that is indistinguishable from human graders (Landauer et al., 1997).

An automated reading tutor helps improve literacy by having children read stories and using a speech recognizer to intervene when the reader asks for reading help or makes mistakes (Mostow and Aist, 1999).

A computer equipped with a vision system watches a short video clip of a soccer match and provides an automated natural language report on the game (Wahlster, 1989).

A computer predicts upcoming words or expands telegraphic speech to assist people with a speech or communication disability (Newell et al., 1998; McCoy et al., 1998).

1.6 Some Brief History

Historically, speech and language processing has been treated very differently in computer science, electrical engineering, linguistics, and psychology/cognitive science. Because of this diversity, speech and language processing encompasses a number of different but overlapping fields in these different departments: **computational linguistics** in linguistics, **natural language processing** in computer science, **speech recognition** in electrical engineering, **computational psycholinguistics** in psychology. This section summarizes the different historical threads which have given rise to the field of speech and language processing. This section will provide only a sketch; see the individual chapters for more detail on each area and its terminology.

Foundational Insights: 1940s and 1950s

The earliest roots of the field date to the intellectually fertile period just after World War II that gave rise to the computer itself. This period from the 1940s through the end of the 1950s saw intense work on two foundational paradigms: the **automaton** and **probabilistic** or **information-theoretic models**.

The automaton arose in the 1950s out of Turing's (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science. Turing's work led first to the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the neuron as a kind of

computing element that could be described in terms of propositional logic, and then to the work of Kleene (1951) and (1956) on finite automata and regular expressions. Shannon (1948) applied probabilistic models of discrete Markov processes to automata for language. Drawing the idea of a finite-state Markov process from Shannon's work, Chomsky (1956) first considered finite-state machines as a way to characterize a grammar, and defined a finite-state language as a language generated by a finite-state grammar. These early models led to the field of **formal language theory**, which used algebra and set theory to define formal languages as sequences of symbols. This includes the context-free grammar, first defined by Chomsky (1956) for natural languages but independently discovered by Backus (1959) and Naur et al. (1960) in their descriptions of the ALGOL programming language.

The second foundational insight of this period was the development of probabilistic algorithms for speech and language processing, which dates to Shannon's other contribution: the metaphor of the **noisy channel** and **decoding** for the transmission of language through media like communication channels and speech acoustics. Shannon also borrowed the concept of **entropy** from thermodynamics as a way of measuring the information capacity of a channel, or the information content of a language, and performed the first measure of the entropy of English using probabilistic techniques.

It was also during this early period that the sound spectrograph was developed (Koenig et al., 1946), and foundational research was done in instrumental phonetics that laid the groundwork for later work in speech recognition. This led to the first machine speech recognizers in the early 1950s. In 1952, researchers at Bell Labs built a statistical system that could recognize any of the 10 digits from a single speaker (Davis et al., 1952). The system had 10 speaker-dependent stored patterns roughly representing the first two vowel formants in the digits. They achieved 97–99% accuracy by choosing the pattern which had the highest relative correlation coefficient with the input.

The Two Camps: 1957–1970

By the end of the 1950s and the early 1960s, speech and language processing had split very cleanly into two paradigms: symbolic and stochastic.

The symbolic paradigm took off from two lines of research. The first was the work of Chomsky and others on formal language theory and generative syntax throughout the late 1950s and early to mid 1960s, and the work of many linguistics and computer scientists on parsing algorithms, initially top-down and bottom-up and then via dynamic programming. One of the earliest

complete parsing systems was Zelig Harris's Transformations and Discourse Analysis Project (TDAP), which was implemented between June 1958 and July 1959 at the University of Pennsylvania (Harris, 1962).[2] The second line of research was the new field of artificial intelligence. In the summer of 1956 John McCarthy, Marvin Minsky, Claude Shannon, and Nathaniel Rochester brought together a group of researchers for a two-month workshop on what they decided to call artificial intelligence (AI). Although AI always included a minority of researchers focusing on stochastic and statistical algorithms (include probabilistic models and neural nets), the major focus of the new field was the work on reasoning and logic typified by Newell and Simon's work on the Logic Theorist and the General Problem Solver. At this point early natural language understanding systems were built, These were simple systems that worked in single domains mainly by a combination of pattern matching and keyword search with simple heuristics for reasoning and question-answering. By the late 1960s more formal logical systems were developed.

The stochastic paradigm took hold mainly in departments of statistics and of electrical engineering. By the late 1950s the Bayesian method was beginning to be applied to the problem of optical character recognition. Bledsoe and Browning (1959) built a Bayesian system for text-recognition that used a large dictionary and computed the likelihood of each observed letter sequence given each word in the dictionary by multiplying the likelihoods for each letter. Mosteller and Wallace (1964) applied Bayesian methods to the problem of authorship attribution on *The Federalist* papers.

The 1960s also saw the rise of the first serious testable psychological models of human language processing based on transformational grammar, as well as the first on-line corpora: the Brown corpus of American English, a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.), which was assembled at Brown University in 1963–64 (Kučera and Francis, 1967; Francis, 1979; Francis and Kučera, 1982), and William S. Y. Wang's 1967 DOC (Dictionary on Computer), an on-line Chinese dialect dictionary.

Four Paradigms: 1970–1983

The next period saw an explosion in research in speech and language processing and the development of a number of research paradigms that still dominate the field.

[2] This system was reimplemented recently and is described by Joshi and Hopely (1999) and Karttunen (1999), who note that the parser was essentially implemented as a cascade of finite-state transducers.

The **stochastic** paradigm played a huge role in the development of speech recognition algorithms in this period, particularly the use of the Hidden Markov Model and the metaphors of the noisy channel and decoding, developed independently by Jelinek, Bahl, Mercer, and colleagues at IBM's Thomas J. Watson Research Center, and by Baker at Carnegie Mellon University, who was influenced by the work of Baum and colleagues at the Institute for Defense Analyses in Princeton. AT&T's Bell Laboratories was also a center for work on speech recognition and synthesis; see Rabiner and Juang (1993) for descriptions of the wide range of this work.

The **logic-based** paradigm was begun by the work of Colmerauer and his colleagues on Q-systems and metamorphosis grammars (Colmerauer, 1970, 1975), the forerunners of Prolog, and Definite Clause Grammars (Pereira and Warren, 1980). Independently, Kay's (1979) work on functional grammar, and shortly later, Bresnan and Kaplan's (1982) work on LFG, established the importance of feature structure unification.

The **natural language understanding** field took off during this period, beginning with Terry Winograd's SHRDLU system, which simulated a robot embedded in a world of toy blocks (Winograd, 1972a). The program was able to accept natural language text commands *(Move the red block on top of the smaller green one)* of a hitherto unseen complexity and sophistication. His system was also the first to attempt to build an extensive (for the time) grammar of English, based on Halliday's systemic grammar. Winograd's model made it clear that the problem of parsing was well-enough understood to begin to focus on semantics and discourse models. Roger Schank and his colleagues and students (in what was often referred to as the *Yale School*) built a series of language understanding programs that focused on human conceptual knowledge such as scripts, plans and goals, and human memory organization (Schank and Albelson, 1977; Schank and Riesbeck, 1981; Cullingford, 1981; Wilensky, 1983; Lehnert, 1977). This work often used network-based semantics (Quillian, 1968; Norman and Rumelhart, 1975; Schank, 1972; Wilks, 1975c, 1975b; Kintsch, 1974) and began to incorporate Fillmore's notion of case roles (Fillmore, 1968) into their representations (Simmons, 1973).

The logic-based and natural-language understanding paradigms were unified on systems that used predicate logic as a semantic representation, such as the LUNAR question-answering system (Woods, 1967, 1973).

The **discourse modeling** paradigm focused on four key areas in discourse. Grosz and her colleagues introduced the study of substructure in discourse, and of discourse focus (Grosz, 1977a; Sidner, 1983), a number of

researchers began to work on automatic reference resolution (Hobbs, 1978), and the **BDI** (Belief-Desire-Intention) framework for logic-based work on speech acts was developed (Perrault and Allen, 1980; Cohen and Perrault, 1979).

Empiricism and Finite State Models Redux: 1983–1993

This next decade saw the return of two classes of models which had lost popularity in the late 1950s and early 1960s, partially due to theoretical arguments against them such as Chomsky's influential review of Skinner's *Verbal Behavior* (Chomsky, 1959b). The first class was finite-state models, which began to receive attention again after work on finite-state phonology and morphology by Kaplan and Kay (1981) and finite-state models of syntax by Church (1980). A large body of work on finite-state models will be described throughout the book.

The second trend in this period was what has been called the "return of empiricism"; most notably here was the rise of probabilistic models throughout speech and language processing, influenced strongly by the work at the IBM Thomas J. Watson Research Center on probabilistic models of speech recognition. These probabilistic methods and other such data-driven approaches spread into part-of-speech tagging, parsing and attachment ambiguities, and connectionist approaches from speech recognition to semantics.

This period also saw considerable work on natural language generation.

The Field Comes Together: 1994–1999

By the last five years of the millennium it was clear that the field was vastly changing. First, probabilistic and data-driven models had become quite standard throughout natural language processing. Algorithms for parsing, part-of-speech tagging, reference resolution, and discourse processing all began to incorporate probabilities, and employ evaluation methodologies borrowed from speech recognition and information retrieval. Second, the increases in the speed and memory of computers had allowed commercial exploitation of a number of subareas of speech and language processing, in particular speech recognition and spelling and grammar checking. Speech and language processing algorithms began to be applied to Augmentative and Alternative Communication (AAC). Finally, the rise of the Web emphasized the need for language-based information retrieval and information extraction.

On Multiple Discoveries

Even in this brief historical overview, we have mentioned a number of cases of multiple independent discoveries of the same idea. Just a few of the "multiples" to be discussed in this book include the application of dynamic programming to sequence comparison by Viterbi, Vintsyuk, Needleman and Wunsch, Sakoe and Chiba, Sankoff, Reichert *et al.*, and Wagner and Fischer (Chapters 5 and 7); the HMM/noisy channel model of speech recognition by Baker and by Jelinek, Bahl, and Mercer (Chapter 7); the development of context-free grammars by Chomsky and by Backus and Naur (Chapter 9); the proof that Swiss-German has a non-context-free syntax by Huybregts and by Shieber (Chapter 13); the application of unification to language processing by Colmerauer *et al.* and by Kay in (Chapter 11).

Are these multiples to be considered astonishing coincidences? A well-known hypothesis by sociologist of science Robert K. Merton (1961) argues, quite the contrary, that

> all scientific discoveries are in principle multiples, including those that on the surface appear to be singletons.

Of course there are many well-known cases of multiple discovery or invention; just a few examples from an extensive list in Ogburn and Thomas (1922) include the multiple invention of the calculus by Leibnitz and by Newton, the multiple development of the theory of natural selection by Wallace and by Darwin, and the multiple invention of the telephone by Gray and Bell.[3] But Merton gives an further array of evidence for the hypothesis that multiple discovery is the rule rather than the exception, including many cases of putative singletons that turn out be a rediscovery of previously unpublished or perhaps inaccessible work. An even stronger piece of evidence is his ethnomethodological point that scientists themselves act under the assumption that multiple invention is the norm. Thus many aspects of scientific life are designed to help scientists avoid being "scooped"; submission dates on journal articles; careful dates in research records; circulation of preliminary or technical reports.

[3] Ogburn and Thomas are generally credited with noticing that the prevalence of multiple inventions suggests that the cultural milieu and not individual genius is the deciding causal factor in scientific discovery. In an amusing bit of recursion, however, Merton notes that even this idea has been multiply discovered, citing sources from the 19th century and earlier!

A Final Brief Note on Psychology

Many of the chapters in this book include short summaries of psychological research on human processing. Of course, understanding human language processing is an important scientific goal in its own right and is part of the general field of cognitive science. However, an understanding of human language processing can often be helpful in building better machine models of language. This seems contrary to the popular wisdom, which holds that direct mimicry of nature's algorithms is rarely useful in engineering applications. For example, the argument is often made that if we copied nature exactly, airplanes would flap their wings; yet airplanes with fixed wings are a more successful engineering solution. But language is not aeronautics. Cribbing from nature is sometimes useful for aeronautics (after all, airplanes do have wings), but it is particularly useful when we are trying to solve human-centered tasks. Airplane flight has different goals than bird flight; but the goal of speech recognition systems, for example, is to perform exactly the task that human court reporters perform every day: transcribe spoken dialog. Since people already do this well, we can learn from nature's previous solution. Since an important application of speech and language processing systems is for human-computer interaction, it makes sense to copy a solution that behaves the way people are accustomed to.

1.7 SUMMARY

This chapter introduces the field of speech and language processing. The following are some of the highlights of this chapter.

- A good way to understand the concerns of speech and language processing research is to consider what it would take to create an intelligent agent like HAL from 2001: A Space Odyssey.
- Speech and language technology relies on formal models, or representations, of knowledge of language at the levels of phonology and phonetics, morphology, syntax, semantics, pragmatics and discourse. A small number of formal models including state machines, formal rule systems, logic, and probability theory are used to capture this knowledge.
- The foundations of speech and language technology lie in computer science, linguistics, mathematics, electrical engineering and psychology. A small number of algorithms from standard frameworks are used

throughout speech and language processing,

- The critical connection between language and thought has placed speech and language processing technology at the center of debate over intelligent machines. Furthermore, research on how people interact with complex media indicates that speech and language processing technology will be critical in the development of future technologies.
- Revolutionary applications of speech and language processing are currently in use around the world. Recent advances in speech recognition and the creation of the World-Wide Web will lead to many more applications.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Research in the various subareas of speech and language processing is spread across a wide number of conference proceedings and journals. The conferences and journals most centrally concerned with computational linguistics and natural language processing are associated with the Association for Computational Linguistics (ACL), its European counterpart (EACL), and the International Conference on Computational Linguistics (COLING). The annual proceedings of ACL and EACL, and the biennial COLING conference are the primary forums for work in this area. Related conferences include the biennial conference on Applied Natural Language Processing (ANLP) and the conference on Empirical Methods in Natural Language Processing (EMNLaP). The journal *Computational Linguistics* is the premier publication in the field, although it has a decidedly theoretical and linguistic orientation. The journal *Natural Language Engineering* covers more practical applications of speech and language research.

Research on speech recognition, understanding, and synthesis is presented at the biennial International Conference on Spoken Language Processing (ICSLP) which alternates with the European Conference on Speech Communication and Technology (EUROSPEECH). The IEEE International Conference on Acoustics, Speech, and Signal Processing (IEEE ICASSP) is held annually, as is the meeting of the Acoustical Society of America. Speech journals include *Speech Communication*, *Computer Speech and Language*, and the *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Work on language processing from an Artificial Intelligence perspective can be found in the annual meetings of the American Association for Artificial Intelligence (AAAI), as well as the biennial International Joint Conference on Artificial Intelligence (IJCAI) meetings. The following artificial intelligence publications periodically feature work on speech and language processing: *Artificial Intelligence*, *Computational Intelligence*, *IEEE Transactions on Intelligent Systems*, and the *Journal of Artificial Intelligence Research*. Work on cognitive modeling of language can be found at the annual meeting of the Cognitive Science Society, as well as its journal *Cognitive Science*. An influential series of invitation-only workshops was held by ARPA, called variously the *DARPA Speech and Natural Language Processing Workshop* or the *ARPA Workshop on Human Language Technology*.

There are a fair number of textbooks available covering various aspects of speech and language processing. Manning and Schütze (1999) (*Foundations of Statistical Language Processing*) focuses on statistical models of tagging, parsing, disambiguation, collocations, and other areas. Charniak (1993) (*Statistical Language Learning*) is an accessible, though older and less-extensive, introduction to similar material. Allen (1995) (*Natural Language Understanding*) provides extensive coverage of language processing from the AI perspective. Gazdar and Mellish (1989) (*Natural Language Processing in Lisp/Prolog*) covers especially automata, parsing, features, and unification. Pereira and Shieber (1987) gives a Prolog-based introduction to parsing and interpretation. Russell and Norvig (1995) is an introduction to artificial intelligence that includes chapters on natural language processing. Partee et al. (1990) has a very broad coverage of mathematical linguistics. Cole (1997) is a volume of survey papers covering the entire field of speech and language processing. A somewhat dated but still tremendously useful collection of foundational papers can be found in Grosz et al. (1986) (*Readings in Natural Language Processing*).

Of course, a wide-variety of speech and language processing resources are now available on the World-Wide Web. Pointers to these resources are maintained on the home-page for this book at:

`http://www.cs.colorado.edu/~martin/slp.html`.

Part I

WORDS

Words are the fundamental building block of language. Every human language, spoken, signed, or written, is composed of words. Every area of speech and language processing, from speech recognition to machine translation to information retrieval on the Web, requires extensive knowledge about words. Psycholinguistic models of human language processing and models from generative linguistics are also heavily based on lexical knowledge.

The six chapters in this part introduce computational models of the spelling, pronunciation, and morphology of words and cover three important real-world tasks that rely on lexical knowledge: automatic speech recognition (ASR), text-to-speech synthesis (TTS), and the correction of spelling errors. Finally, these chapters define perhaps the most important computational model for speech and language processing: the automaton. Four kinds of automata are covered: finite-state automata (FSAs) and regular expressions, finite-state transducers (FSTs), weighted transducers, and the Hidden Markov Model (HMM), as well as the N-gram model of word sequences.

2 REGULAR EXPRESSIONS AND AUTOMATA

In the old days, if you wanted to impeach a witness you had to go back and fumble through endless transcripts. Now it's on a screen somewhere or on a disk and I can search for a particular word — say every time the witness used the word glove *— and then quickly ask a question about what he said years ago. Right away you see the witness get flustered.*

Johnnie L. Cochran Jr., attorney, *New York Times*, 9/28/97

Imagine that you have become a passionate fan of woodchucks. Desiring more information on this celebrated woodland creature, you turn to your favorite Web browser and type in *woodchuck*. Your browser returns a few sites. You have a flash of inspiration and type in *woodchucks*. This time you discover "interesting links to woodchucks and lemurs" and "all about Vermont's unique, endangered species". Instead of having to do this search twice, you would have rather typed one search command specifying something like *woodchuck with an optional final s*. Furthermore, you might want to find a site whether or not it spelled *woodchucks* with a capital *W* (*Woodchuck*). Or perhaps you might want to search for all the prices in some document; you might want to see all strings that look like *$199* or *$25* or *$24.99*. In this chapter we introduce the **regular expression**, the standard notation for characterizing text sequences. The regular expression is used for specifying text strings in situations like this Web-search example, and in other information retrieval applications, but also plays an important role in word-processing (in PC, Mac, or UNIX applications), computation of frequencies from corpora, and other such tasks.

After we have defined regular expressions, we show how they can be implemented via the **finite-state automaton**. The finite-state automaton is not only the mathematical device used to implement regular expressions, but

also one of the most significant tools of computational linguistics. Variations of automata such as finite-state transducers, Hidden Markov Models, and N-gram grammars are important components of the speech recognition and synthesis, spell-checking, and information-extraction applications that we will introduce in later chapters.

2.1 Regular Expressions

SIR ANDREW: Her C's, her U's and her T's: why that?
Shakespeare, *Twelfth Night*

REGULAR EXPRESSION

One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search strings. The regular expression languages used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect are almost identical, and many RE features exist in the various Web search engines. Besides this practical use, the regular expression is an important theoretical tool throughout computer science and linguistics.

STRINGS

A regular expression (first developed by Kleene (1956) but see the History section for more details) is a formula in a special language that is used for specifying simple classes of **strings**. A string is a sequence of symbols; for the purpose of most text-based search techniques, a string is any sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation). For these purposes a space is just a character like any other, and we represent it with the symbol ␣.

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus they can be used to specify search strings as well as to define a language in a formal way. We will begin by talking about regular expressions as a way of specifying searches in texts, and proceed to other uses. Section 2.3 shows that the use of just three regular expression operators is sufficient to characterize strings, but we use the more convenient and commonly-used regular expression syntax of the Perl language throughout this section. Since common text-processing programs agree on most of the syntax of regular expressions, most of what we say extends to all UNIX, Microsoft Word, and WordPerfect regular expressions. Appendix A shows the few areas where these programs differ from the Perl syntax.

CORPUS

Regular expression search requires a **pattern** that we want to search for, and a **corpus** of texts to search through. A regular expression search

function will search through the corpus returning all texts that contain the pattern. In an information retrieval (IR) system such as a Web search engine, the texts might be entire documents or Web pages. In a word-processor, the texts might be individual words, or lines of a document. In the rest of this chapter, we will use this last paradigm. Thus when we give a search pattern, we will assume that the search engine returns the *line of the document* returned. This is what the UNIX `grep` command does. We will underline the exact part of the pattern that matches the regular expression. A search can be designed to return all matches to a regular expression or only the first match. We will show only the first match.

Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters. For example, to search for *woodchuck*, we type `/woodchuck/`. So the regular expression `/Buttercup/` matches any string containing the substring *Buttercup*, for example the line *I'm called little Buttercup*) (recall that we are assuming a search application that returns entire lines). From here on we will put slashes around each regular expression to make it clear what is a regular expression and what is a pattern. We use the slash since this is the notation used by Perl, but the slashes are *not* part of the regular expressions.

The search string can consist of a single letter (like `/!/`) or a sequence of letters (like `/urgl/`); The *first* instance of each match to the regular expression is underlined below (although a given application might choose to return more than just the first instance):

RE	Example Patterns Matched
`/woodchucks/`	"interesting links to woodchucks and lemurs"
`/a/`	"Mary Ann stopped by Mona's"
`/Claire_says,/`	"Dagmar, my gift please," Claire says,"
`/song/`	"all our pretty songs"
`/!/`	"You've left the burglar behind again!" said Nori

Regular expressions are **case sensitive**; lowercase `/s/` is distinct from uppercase `/S/`; (`/s/` matches a lower case *s* but not an uppercase *S*). This means that the pattern `/woodchucks/` will not match the string *Woodchucks*. We can solve this problem with the use of the square braces `[` and `]`. The string of characters inside the braces specify a **disjunction** of characters to match. For example Figure 2.1 shows that the pattern `/[wW]/` matches patterns containing either *w* or *W*.

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Figure 2.1 The use of the brackets [] to specify a disjunction of characters.

The regular expression /[1234567890]/ specified any single digit. While classes of characters like digits or letters are important building blocks in expressions, they can get awkward (e.g., it's inconvenient to specify

/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/

to mean "any capital letter"). In these cases the brackets can be used with the dash (-) to specify any one character in a **range**. The pattern /[2-5]/ specifies any one of the characters *2*, *3*, *4*, or *5*. The pattern /[b-g]/ specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples:

RANGE

RE	Match	Example Patterns Matched
/[A-Z]/	an uppercase letter	"we should call it 'Drenched Blossoms'"
/[a-z]/	a lowercase letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.2 The use of the brackets [] plus the dash - to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ^. If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated. For example, the pattern /[^a]/ matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Figure 2.3 shows some examples.

RE	Match (single characters)	Example Patterns Matched
[^A-Z]	not an uppercase letter	"Oyfn pripetchik"
[^Ss]	neither 'S' nor 's'	"I have no exquisite reason for't"
[^\.]	not a period	"our resident Djinn"
[e^]	either 'e' or '^'	"look up ^ now"
a^b	the pattern 'a^b'	"look up a^ b now"

Figure 2.3 Uses of the caret ^ for negation or just to mean ^ .

The use of square braces solves our capitalization problem for *woodchucks*. But we still haven't answered our original question; how do we specify both *woodchuck* and *woodchucks*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question-mark `/?/`, which means "the preceding character or nothing", as shown in Figure 2.4.

RE	Match	Example Patterns Matched
`woodchucks?`	woodchuck or woodchucks	"woodchuck"
`colou?r`	color or colour	"colour"

Figure 2.4 The question-mark ? marks optionality of the previous expression.

We can think of the question-mark as meaning "zero or one instances of the previous character". That is, it's a way of specifying how many of something that we want. So far we haven't needn't to specify that we want more than one of something. But sometimes we need regular expressions that allow repetitions of things. For example, consider the language of (certain) sheep, which consists of strings that look like the following:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
...

This language consists of strings with a *b*, followed by at least two *a*s, followed by an exclamation point. The set of operators that allow us to say things like "some number of *a*s" are based on the asterisk or `*`, commonly called the **Kleene** * (pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `/a*/` means "any string of zero or more *a*s". This will match *a* or *aaaaaa* but it will also match *Off Minor*, since the string *Off Minor* has zero *a*s. So the regular expression for matching one or more *a* is `/aa*/`, meaning one *a* followed by zero or more *a*s. More complex patterns can also be repeated. So `/[ab]*/` means "zero or more *a*s or *b*s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*. KLEENE *

We now know enough to specify part of our regular expression for prices: multiple digits. Recall that the regular expression for an individual digit was `/[0-9]/`. So the regular expression for an integer (a string of digits) is `/[0-9][0-9]*/`. (Why isn't it just `/[0-9]*/`)?

Sometimes it's annoying to have to write the regular expression for digits twice, so there is a shorter way to specify "at least one" of some character.

KLEENE + This is the **Kleene +**, which means "one or more of the previous character". Thus the expression `/[0-9]+/` is the normal way to specify "a sequence of digits". There are thus two ways to specify the sheep language: `/baaa*!/` or `/baa+!/`.

One very important special character is the period (`/./`, a **wildcard** expression that matches any single character (*except* a carriage return):

RE	Match	Example Patterns
`/beg.n/`	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.5 The use of the period . to specify any character.

The wildcard is often used together with the Kleene star to mean "any string of characters". For example suppose we want to find any line in which a particular word, for example *aardvark*, appears twice. We can specify this with the regular expression `/aardvark.*aardvark/`.

ANCHORS **Anchors** are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret ^ and the dollar-sign $. The caret ^ matches the start of a line. The pattern `/^The/` matches the word *The* only at the start of a line. Thus there are three uses of the caret ^: to match the start of a line, as a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow Perl to know which function a given caret is supposed to have?). The dollar sign $ matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `/^The dog\.$/` matches a line that contains only the phrase *The dog*. (We have to use the backslash here since we want the . to mean "period" and not the wildcard).

There are also two other anchors: `\b` matches a word boundary, while `\B` matches a non-boundary. Thus `/\bthe\b/` matches the word *the* but not the word *other*. More technically, Perl defines a word as any sequence of digits, underscores or letters; this is based on the definition of "words" in programming languages like Perl or C. For example, `/\b99/` will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows

a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case we might want to search for either the string *cat* or the string *dog*. Since we can't use the square-brackets to search for "cat or dog" (why not?) we need a new operator, the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

DISJUNCTION

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. In order to make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene`*`. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

PRECEDENCE

The parenthesis operator `(` is also useful when we are using counters like the Kleene`*`. Unlike the `|` operator, the Kleene`*` operator applies by default only to a single character, not a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression `/Column␣[0-9]+␣*/` will not match any column; instead, it will match a column followed by any number of spaces! The star here applies only to the space `␣` that precedes it, not the whole sequence. With the parentheses, we could write the expression `/(Column␣[0-9]+␣*)*/` to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated any number of times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order of RE operator precedence, from highest precedence to lowest precedence:

OPERATOR PRECEDENCE

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

Thus, because counters have a higher precedence than sequences, `/the*/` matches *theeeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `/the|any/` matches *the* or *any* but not *theny*.

Patterns can be ambiguous in another way. Consider the expression `/[a-z]*/` when matching against the text *once upon a time*. Since `/[a-z]*/` matches zero or more letters, this expression could match nothing, or just the first letter *o*, or *on*, or *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

GREEDY

A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
/the/
```

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
/[tT]he/
```

But we will still incorrectly return texts with `the` embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word boundary on both sides:

```
/\b[tT]he\b/
```

Suppose we wanted to do this without the use of `/\b/`? We might want this since `/\b/` won't treat underscores and numbers as word boundaries; but we might want to find *the* in some context where it might also have underlines or numbers nearby (*the_* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

```
/[^a-zA-Z][tT]he[^a-zA-Z]/
```

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression `[^a-zA-Z]`, which we used to avoid embedded *thes*, implies that there must be some single (although non-alphabetic) character before the `the`. We can avoid this by specifying that before the *the* we require *either* the beginning-of-line or a non-alphabetic character:

```
/(^|[^a-zA-Z])[tT]he[^a-zA-Z]/
```

A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The user might want "any PC with more than 500 MHz and 32 Gb of disk space for less than $1000". In order to do this kind of retrieval we will first need to be able to look for expressions like *500 MHz* or *32 Gb* or *Compaq* or *Mac* or *$999.99*. In the rest of this section we'll work out some simple regular expressions for this task.

First, let's complete our regular expression for prices. Here's a regular expression for a dollar sign followed by a string of digits. Note that Perl is smart enough to realize that `$` here doesn't mean end-of-line; how might it know that?

```
/$[0-9]+/
```

Now we just need to deal with fractions of dollars. We'll add a decimal point and two digits afterwards:

```
/$[0-9]+\.[0-9][0-9]/
```

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional, and make sure we're at a word boundary:

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

How about specifications for processor speed (in megahertz = MHz or gigahertz = GHz)? Here's a pattern for that:

```
/\b[0-9]+␣*(MHz|[Mm]egahertz|GHz|[Gg]igahertz)\b/
```

Note that we use `/␣*/` to mean "zero or more spaces", since there might always be extra spaces lying around. Dealing with disk space (in Gb = gigabytes), or memory size (in Mb = megabytes or Gb = gigabytes), we

need to allow for optional gigabyte fractions again (*5.5 Gb*). Note the use of `?` for making the final `s` optional:

```
/\b[0-9]+␣*(Mb|[Mm]egabytes?)\b/
/\b[0-9](\.[0-9]+)?␣*(Gb|[Gg]igabytes?)\b/
```

Finally, we might want some simple patterns to specify operating systems and vendors:

```
/\b(Win95|Win98|WinNT|Windows␣*(NT|95|98|2000)?)\b/
/\b(Mac|Macintosh|Apple)\b/
```

Advanced Operators

RE	Expansion	Match	Example Patterns
`\d`	`[0-9]`	any digit	Party␣of␣5
`\D`	`[^0-9]`	any non-digit	Blue␣moon
`\w`	`[a-zA-Z0-9␣]`	any alphanumeric or space	Daiyu
`\W`	`[^\w]`	a non-alphanumeric	!!!!
`\s`	`[␣\r\t\n\f]`	whitespace (space, tab)	
`\S`	`[^\s]`	Non-whitespace	in␣Concord

Figure 2.6 Aliases for common sets of characters.

There are also some useful advanced regular expression operators. Figure 2.6 shows some useful aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene +, we can also use explicit numbers as counters, by enclosing them in curly brackets. The regular expression `/{3}/` means "exactly 3 occurrences of the previous character or expression". So `/a\.{24}z/` will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a z).

A range of numbers can also be specified; so `/{n,m}/` specifies from n to m occurrences of the previous char or expression, while `/{n,}/` means at least n occurrences of the previous expression. REs for counting are summarized in Figure 2.7.

NEWLINE

Finally, certain special characters are referred to by special notation based on the backslash (\). The most common of these are the **newline** character `\n` and the **tab** character `\t`. To refer to characters that are special themselves, (like `.`, `*`, `[`, and `\`), precede them with a backslash, (i.e., `/\./`, `/\*/`, `/\[/`, and `/\\/`).

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression

Figure 2.7 Regular expression operators for counting.

RE	Match	Example Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Would you light my candle?"
\n	a newline	
\t	a tab	

Figure 2.8 Some characters that need to be backslashed.

The reader should consult Appendix A for further details of regular expressions, and especially for the differences between regular expressions in Perl, UNIX, and Microsoft Word.

Regular Expression Substitution, Memory, and ELIZA

An important use of regular expressions is in **substitutions**. For example, the Perl substitution operator `s/regexp1/regexp2/` allows a string characterized by one regular expression to be replaced by a string characterized by a different regular expression: SUBSTITUTION

```
s/colour/color/
```

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all integers in a text, changing e.g., *the 35 boxes* to *the <35> boxes*. We'd like a way to refer back to the integer we've found so that we can easily add the brackets. To do this, we put parentheses (and) around the first pattern, and use the **number** operator `\1` in the second pattern to refer back. Here's how it looks:

```
s/([0-9]+)/<\1>/
```

The parenthesis and number operators can also be used to specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern "the Xer they were, the Xer they will be", where we want to constrain the two X's to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator `\1`, as follows:

```
/the (.*)er they were, the \1er they will be/
```

Here the `\1` will be replaced by whatever string matched the first item in parentheses. So this will match *The bigger they were, the bigger they will be* but not *The bigger they were, the faster they will be*.

The number operator can be used with other numbers: if you match two different sets of parenthesis, `\2` means whatever matched the *second* set. For example

```
/the (.*)er they (.*), the \1er they \2/
```

will match *The bigger they were, the bigger they were* but not *The bigger they were, the bigger they will be*. These numbered memories are called **registers** (e.g. register 1, register 2, register 3, etc). This memory feature is not part of every regular expression language and is often considered an "extended" feature of regular expressions.

REGISTERS

Substitutions using memory are very useful in implementing a simple natural-language understanding program like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulated a Rogerian psychologist and could carry on conversations with the user like the following:

User_1:	Men are all alike.
ELIZA_1:	IN WHAT WAY
User_2:	They're always bugging us about something or other.
ELIZA_2:	CAN YOU THINK OF A SPECIFIC EXAMPLE
User_3:	Well, my boyfriend made me come here.
ELIZA_3:	YOUR BOYFRIEND MADE YOU COME HERE
User_4:	He says I'm depressed much of the time.
ELIZA_4:	I AM SORRY TO HEAR YOU ARE DEPRESSED.

Eliza worked by having a cascade of regular expression substitutions that each matched some part of the input lines and changed them. The first substitutions changed all instances of *my* to *YOUR*, and *I'm* to *YOU ARE*, and so on. The next set of substitutions looked for relevant patterns in the input and created an appropriate output; here are some examples:

```
s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
```

```
s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK  OF A SPECIFIC EXAMPLE/
```

Since multiple substitutions could apply to a given input, substitutions were assigned a rank and were applied in order. Creation of such patterns is addressed in Exercise 2.2.

2.2 FINITE-STATE AUTOMATA

The regular expression is more than just a convenient metalanguage for text searching. First, a regular expression is one way of describing a **finite-state automaton** (**FSA**). Finite-state automata are the theoretical foundation of a good deal of the computational work we will describe in this book. Any regular expression can be implemented as a finite-state automaton (except regular expressions that use the memory feature; more on this later). Symmetrically, any finite-state automaton can be described with a regular expression. Second, a regular expression is one way of characterizing a particular kind of formal language called a **regular language**. Both regular expressions and finite-state automata can be used to described regular languages. The relation among these three theoretical constructions is sketched out in Figure 2.9.

FINITE-STATE AUTOMATON

FSA

REGULAR LANGUAGE

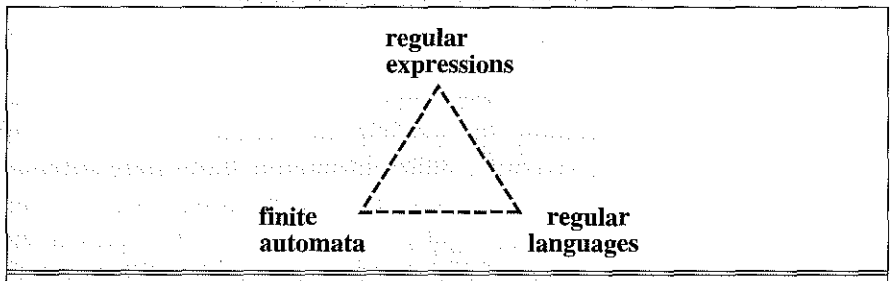


Figure 2.9 The relationship between finite automata, regular expressions, and regular languages; figure suggested by Martin Kay.

This section will begin by introducing finite-state automata for some of the regular expressions from the last section, and then suggest how the mapping from regular expressions to automata proceeds in general. Although we begin with their use for implementing regular expressions, FSAs have a wide variety of other uses that we will explore in this chapter and the next.

Using an FSA to Recognize Sheeptalk

After a while, with the parrot's help, the Doctor got to learn the language of the animals so well that he could talk to them himself and understand everything they said.

Hugh Lofting, *The Story of Doctor Dolittle*

Let's begin with the "sheep language" we discussed previously. Recall that we defined the sheep language as any string from the following (infinite) set:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
...

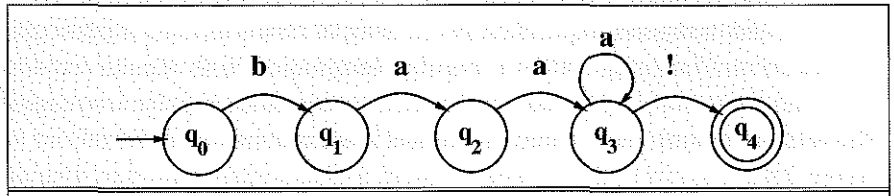


Figure 2.10 A finite-state automaton for talking sheep.

AUTOMATON

The regular expression for this kind of "sheeptalk" is `/baa+!/`. Figure 2.10 shows an **automaton** for modeling this regular expression. The automaton (i.e., machine, also called **finite automaton**, **finite-state automaton**, or **FSA**) recognizes a set of strings, in this case the strings characterizing sheep talk, in the same way that a regular expression does. We represent the automaton as a directed graph: a finite set of vertices (also called nodes), together with a set of directed links between pairs of vertices called arcs. We'll represent vertices with circles and arcs with arrows. The automaton has five **states**, which are represented by nodes in the graph. State 0 is the **start state** which we represent by the incoming arrow. State 4 is the **final state** or **accepting state**, which we represent by the double circle. It also has four **transitions**, which we represent by arcs in the graph.

STATE
START STATE

The FSA can be used for recognizing (we also say **accepting**) strings in the following way. First, think of the input as being written on a long tape

broken up into cells, with one symbol written in each cell of the tape, as in Figure 2.11.

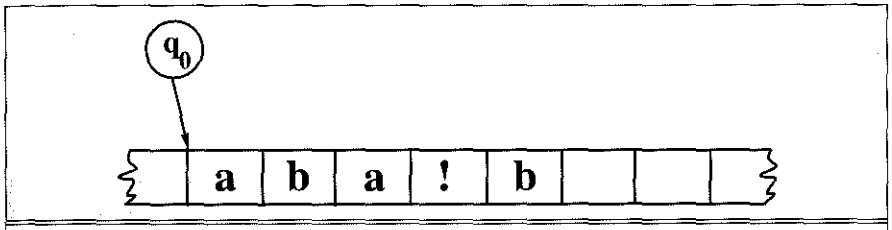


Figure 2.11 A tape with cells.

The machine starts in the start state (q_0), and iterates the following process: Check the next letter of the input. If it matches the symbol on an arc leaving the current state, then cross that arc, move to the next state, and also advance one symbol in the input. If we are in the accepting state (q_4) when we run out of input, the machine has successfully recognized an instance of sheeptalk. If the machine never gets to the final state, either because it runs out of input, or it gets some input that doesn't match an arc (as in Figure 2.11), or if it just happens to get stuck in some non-final state, we say the machine **rejects** or fails to accept an input.

REJECTS

We can also represent an automaton with a **state-transition table**. As in the graph notation, the state-transition table represents the start state, the accepting states, and what transitions leave each state with which symbols. Here's the state-transition table for the FSA of Figure 2.10.

STATE-TRANSITION TABLE

	Input		
State	b	a	!
0	1	0	0
1	0	2	0
2	0	3	0
3	0	3	4
4:	0	0	0

Figure 2.12 The state-transition table for the FSA of Figure 2.10.

We've marked state 4 with a colon to indicate that it's a final state (you can have as many final states as you want), and the 0 indicates an illegal or missing transition. We can read the first row as "if we're in state 0 and we see the input **b** we must go to state 1. If we're in state 0 and we see the input **a** or **!**, we fail".

More formally, a finite automaton is defined by the following five parameters:

- Q: a finite set of N states $q_0, q_1, \ldots, q_N$
- Σ: a finite input alphabet of symbols
- q_0: the start state
- F: the set of final states, $F \subseteq Q$
- $\delta(q,i)$: the transition function or transition matrix between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q,i)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q;

For the sheeptalk automaton in Figure 2.10, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, !\}$, $F = \{q_4\}$, and $\delta(q,i)$ is defined by the transition table in Figure 2.12.

Figure 2.13 presents an algorithm for recognizing a string using a state-transition table. The algorithm is called D-RECOGNIZE for "deterministic recognizer". A **deterministic** algorithm is one that has no choice points; the algorithm always knows what to do for any input. The next section will introduce non-deterministic automata that must make decisions about which states to move to.

DETERMINISTIC

D-RECOGNIZE takes as input a tape and an automaton. It returns *accept* if the string it is pointing to on the tape is accepted by the automaton, and *reject* otherwise. Note that since D-RECOGNIZE assumes it is already pointing at the string to be checked, its task is only a subpart of the general problem that we often use regular expressions for, finding a string in a corpus. (The general problem is left as an exercise to the reader in Exercise 2.9.)

D-RECOGNIZE begins by initializing the variable *index* the beginning of the tape, and *current-state* to the machine's initial state. D-RECOGNIZE then enters a loop that drives the rest of the algorithm. It first checks whether it has reached the end of its input. If so, it either accepts the input (if the current state is an accept state) or rejects the input (if not).

If there is input left on the tape, D-RECOGNIZE looks at the transition table to decide which state to move to. The variable *current-state* indicates which row of the table to consult, while the current symbol on the tape indicates which column of the table to consult. The resulting transition-table cell is used to update the variable *current-state* and *index* is incremented to move forward on the tape. If the transition-table cell is empty then the machine has nowhere to go and must reject the input.

Figure 2.14 traces the execution of this algorithm on the sheep language FSA given the sample input string *baaa!*.

```
function D-RECOGNIZE(tape, machine) returns accept or reject

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
   if End of input has been reached then
    if current-state is an accept state then
      return accept
    else
      return reject
   elsif transition-table[current-state,tape[index]] is empty then
     return reject
   else
     current-state ← transition-table[current-state,tape[index]]
     index ← index + 1
  end
```

Figure 2.13 An algorithm for deterministic recognition of FSAs. This algorithm returns *accept* if the entire string it is pointing at is in the language defined by the FSA, and *reject* if the string is not in the language.

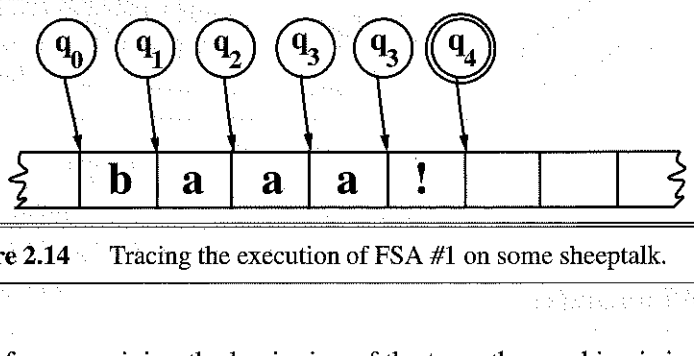


Figure 2.14 Tracing the execution of FSA #1 on some sheeptalk.

Before examining the beginning of the tape, the machine is in state q_0. Finding a *b* on input tape, it changes to state q_1 as indicated by the contents of *transition-table*[q_0,*b*] in Figure 2.12 on page 35. It then finds an *a* and switches to state q_2, another *a* puts it in state q_3, a third *a* leaves it in state q_3, where it reads the "!", and switches to state q_4. Since there is no more input, the `End of input` condition at the beginning of the loop is satisfied for the first time and the machine halts in q_4. State q_4 is an accepting state, and so the machine has accepted the string *baaa!* as a sentence in the sheep language.

The algorithm will fail whenever there is no legal transition for a given combination of state and input. The input *abc* will fail to be recognized since there is no legal transition out of state q_0 on the input a, (i.e., this entry of the transition table in Figure 2.12 on page 35 has a $\emptyset$). Even if the automaton had allowed an initial *a* it would have certainly failed on *c*, since *c* isn't even in the sheeptalk alphabet!. We can think of these "empty" elements in the table as if they all pointed at one "empty" state, which we might call the **fail state** or **sink state**. In a sense then, we could view any machine with empty transitions *as if* we had augmented it with a fail state, and drawn in all the extra arcs, so we always had somewhere to go from any state on any possible input. Just for completeness, Figure 2.15 shows the FSA from Figure 2.10 with the fail state q_F filled in.

FAIL STATE

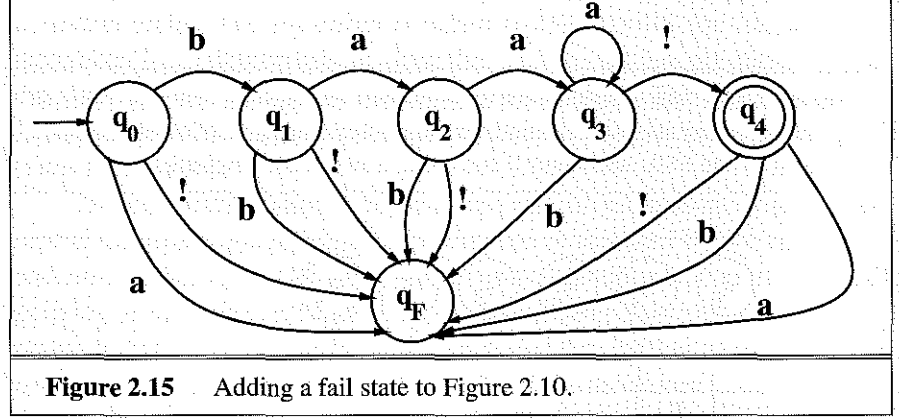


Figure 2.15 Adding a fail state to Figure 2.10.

Formal Languages

We can use the same graph in Figure 2.10 as an automaton for GENERATING sheeptalk. If we do, we would say that the automaton starts at state q_0, and crosses arcs to new states, printing out the symbols that label each arc it follows. When the automaton gets to the final state it stops. Notice that at state 3, the automaton has to chose between printing out a ! and going to state 4, or printing out an **a** and returning to state 3. Let's say for now that we don't care how the machine makes this decision; maybe it flips a coin. For now, we don't care which exact string of sheeptalk we generate, as long as it's a string captured by the regular expression for sheeptalk above.

Key Concept #1. Formal Language: A model which can both generate and recognize all and only the strings of a formal language acts as a *definition* of the formal language.

A **formal language** is a set of strings, each string composed of symbols from a finite symbol-set called an **alphabet** (the same alphabet used above for defining an automaton!). The alphabet for the sheep language is the set $\Sigma = \{a, b, !\}$. Given a model m (such as a particular FSA), we can use $L(m)$ to mean "the formal language characterized by m". So the formal language defined by our sheeptalk automaton m in Figure 2.10 (and Figure 2.12) is the infinite set:

FORMAL LANGUAGE

ALPHABET

$$L(m) = \{baa!, baaa!, baaaa!, baaaaa!, baaaaaa!, \ldots\} \quad (2.1)$$

The usefulness of an automaton for defining a language is that it can express an infinite set (such as this one above) in a closed form. Formal languages are not the same as **natural languages**, which are the kind of languages that real people speak. In fact, a formal language may bear no resemblance at all to a real language (e.g., a formal language can be used to model the different states of a soda machine). But we often use a formal language to model part of a natural language, such as parts of the phonology, morphology, or syntax. The term **generative grammar** is sometimes used in linguistics to mean a grammar of a formal language; the origin of the term is this use of an automaton to define a language by generating all possible strings.

NATURAL LANGUAGES

Another Example

In the previous examples our formal alphabet consisted of letters; but we can also have a higher level alphabet consisting of words. In this way we can write finite-state automata that model facts about word combinations. For example, suppose we wanted to build an FSA that modeled the subpart of English dealing with amounts of money. Such a formal language would model the subset of English consisting of phrases like *ten cents*, *three dollars*, *one dollar thirty-five cents* and so on.

We might break this down by first building just the automaton to account for the numbers from 1 to 99, since we'll need them to deal with cents. Figure 2.16 shows this.

We could now add *cents* and *dollars* to our automaton. Figure 2.17 shows a simple version of this, where we just made two copies of the automaton in Figure 2.16 and appended the words *cents* and *dollars*.

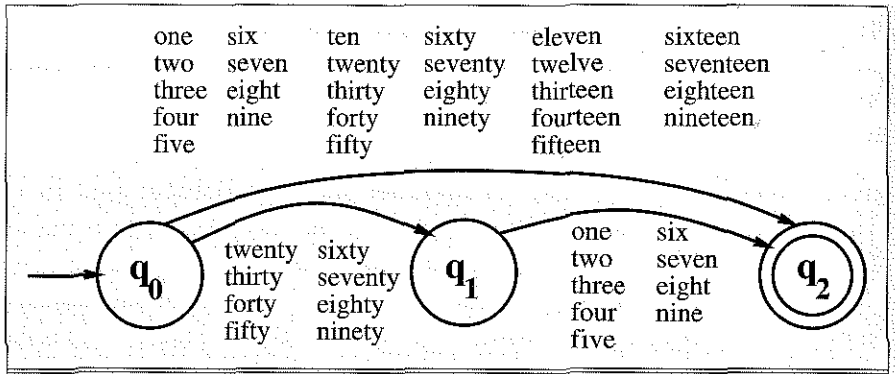


Figure 2.16 An FSA for the words for English numbers 1–99.

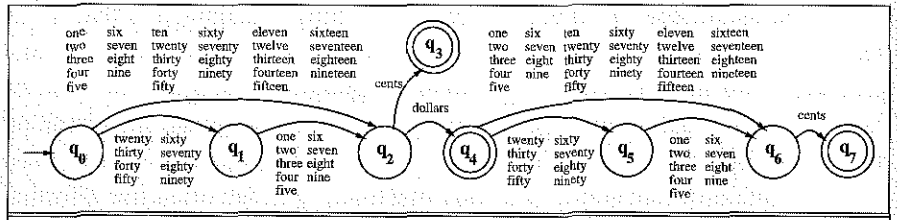


Figure 2.17 FSA for the simple dollars and cents.

We would now need to add in the grammar for different amounts of dollars; including higher numbers like *hundred*, *thousand*. We'd also need to make sure that the nouns like *cents* and *dollars* are singular when appropriate (*one cent*, *one dollar*), and plural when appropriate (*ten cents*, *two dollars*). This is left as an exercise for the reader (Exercise 2.3). We can think of the FSAs in Figure 2.16 and Figure 2.17 as simple grammars of parts of English. We will return to grammar-building in Part II of this book, particularly in Chapter 9.

Non-Deterministic FSAs

Let's extend our discussion now to another class of FSAs: **non-deterministic FSA**s (or **NFSA**s). Consider the sheeptalk automaton in Figure 2.18, which is much like our first automaton in Figure 2.10:

The only difference between this automaton and the previous one is that here in Figure 2.18 the self-loop is on state 2 instead of state 3. Consider using this network as an automaton for recognizing sheeptalk. When we get to state 2, if we see an **a** we don't know whether to remain in state

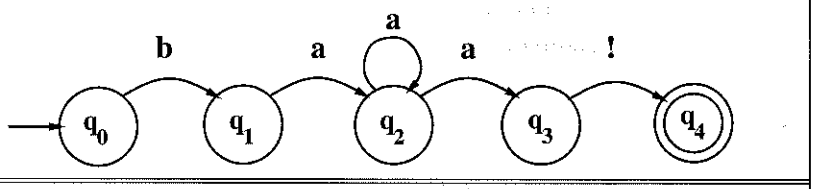


Figure 2.18 A non-deterministic finite-state automaton for talking sheep (NFSA #1). Compare with the deterministic automaton in Figure 2.10.

2 or go on to state 3. Automata with decision points like this are called **non-deterministic FSA**s (or **NFSA**s). Recall by contrast that Figure 2.10 specified a **deterministic** automaton, i.e., one whose behavior during recognition is fully *determined* by the state it is in and the symbol it is looking at. A deterministic automaton can be referred to as a **DFSA**. That is not true for the machine in Figure 2.18 (NFSA #1).

NON-DETERMINISTIC NFSA

DFSA

There is another common type of non-determinism, caused by arcs that have no symbols on them (called **ε-transitions**). The automaton in Figure 2.19 defines the exact same language as the last one, or our first one, but it does it with an ε-transition.

ε-TRANSITION

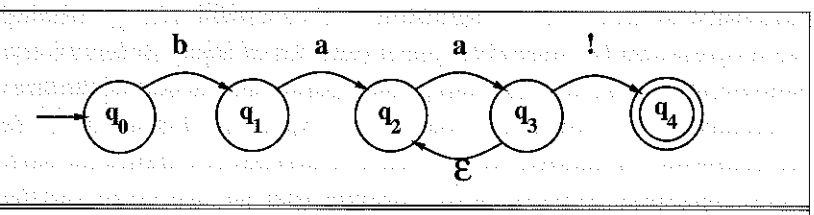


Figure 2.19 Another NFSA for the sheep language (NFSA #2). It differs from NFSA #1 in Figure 2.18 in having an ε-transition.

We interpret this new arc as follows: If we are in state 3, we are allowed to move to state 2 *without* looking at the input, or advancing our input pointer. So this introduces another kind of non-determinism — we might not know whether to follow the ε-transition or the ! arc.

Using an NFSA to Accept Strings

If we want to know whether a string is an instance of sheeptalk or not, and if we use a non-deterministic machine to recognize it, we might follow the wrong arc and reject it when we should have accepted it. That is, since there is more than one choice at some point, we might take the wrong choice. This

problem of choice in non-deterministic models will come up again and again as we build computational models, particularly for parsing.

There are three standard solutions to this problem:

- **Backup:** Whenever we come to a choice point, we could put a *marker* to mark where we were in the input, and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path.
- **Look-ahead:** We could look ahead in the input to help us decide which path to take.
- **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

We will focus here on the backup approach and defer discussion of the look-ahead and parallelism approaches to later chapters.

The backup approach suggests that we should blithely make choices that might lead to deadends, knowing that we can always return to unexplored alternative choices. There are two keys to this approach: we need to remember all the alternatives for each choice point, and we need to store sufficient information about each alternative so that we can return to it when necessary. When a backup algorithm reaches a point in its processing where no progress can be made (because it runs out of input, or has no legal transitions), it returns to a previous choice point, selects one of the unexplored alternatives, and continues from there. Applying this notion to our non-deterministic recognizer, we need only remember two things for each choice point: the state, or node, of the machine that we can go to and the corresponding position on the tape. We will call the combination of the node and position the **search-state** of the recognition algorithm. To avoid confusion, we will refer to the state of the automaton (as opposed to the state of the search) as a **node** or a **machine-state**. Figure 2.21 presents a recognition algorithm based on this approach.

SEARCH-STATE

Before going on to describe the main part of this algorithm, we should note two changes to the transition table that drives it. First, in order to represent nodes that have outgoing ε-transitions, we add a new **ε-column** to the transition table. If a node has an ε-transition, we list the destination node in the ε-column for that node's row. The second addition is needed to account for multiple transitions to different nodes from the same input symbol. We let each cell entry consist of a list of destination nodes rather than a single node. Figure 2.20 shows the transition table for the machine in Figure 2.18 (NFSA #1). While it has no ε-transitions, it does show that in machine-state

	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Figure 2.20 The transition table from NFSA #1 in Figure 2.18.

q_2 the input a can lead back to q_2 or on to q_3.

Figure 2.21 shows the algorithm for using a non-deterministic FSA to recognize an input string. The function ND-RECOGNIZE uses the variable *agenda* to keep track of all the currently unexplored choices generated during the course of processing. Each choice (search state) is a tuple consisting of a node (state) of the machine and a position on the tape. The variable *current-search-state* represents the branch choice being currently explored.

ND-RECOGNIZE begins by creating an initial search-state and placing it on the agenda. For now we don't specify what order the search-states are placed on the agenda. This search-state consists of the initial machine-state of the machine and a pointer to the beginning of the tape. The function NEXT is then called to retrieve an item from the agenda and assign it to the variable *current-search-state*.

As with D-RECOGNIZE, the first task of the main loop is to determine if the entire contents of the tape have been successfully recognized. This is done via a call to ACCEPT-STATE?, which returns *accept* if the current search-state contains both an accepting machine-state and a pointer to the end of the tape. If we're not done, the machine generates a set of possible next steps by calling GENERATE-NEW-STATES, which creates search-states for any ε-transitions and any normal input-symbol transitions from the transition table. All of these search-state tuples are then added to the current agenda.

Finally, we attempt to get a new search-state to process from the agenda. If the agenda is empty we've run out of options and have to reject the input. Otherwise, an unexplored option is selected and the loop continues.

It is important to understand why ND-RECOGNIZE returns a value of reject only when the agenda is found to be empty. Unlike D-RECOGNIZE, it does not return reject when it reaches the end of the tape in an non-accept machine-state or when it finds itself unable to advance the tape from some

machine-state. This is because, in the non-deterministic case, such roadblocks only indicate failure down a given path, not overall failure. We can only be sure we can reject a string when all possible choices have been examined and found lacking.

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

  agenda ← {(Initial state of machine, beginning of tape)}
  current-search-state ← NEXT(agenda)
  loop
    if ACCEPT-STATE?(current-search-state) returns true then
      return accept
    else
      agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

  current-node ← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
    (transition-table[current-node,ε], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state of machine
then
    return true
  else
    return false
```

Figure 2.21 An algorithm for NFSA recognition. The word *node* means a state of the FSA, while *state* or *search-state* means "the state of the search process", i.e., a combination of *node* and *tape-position*.

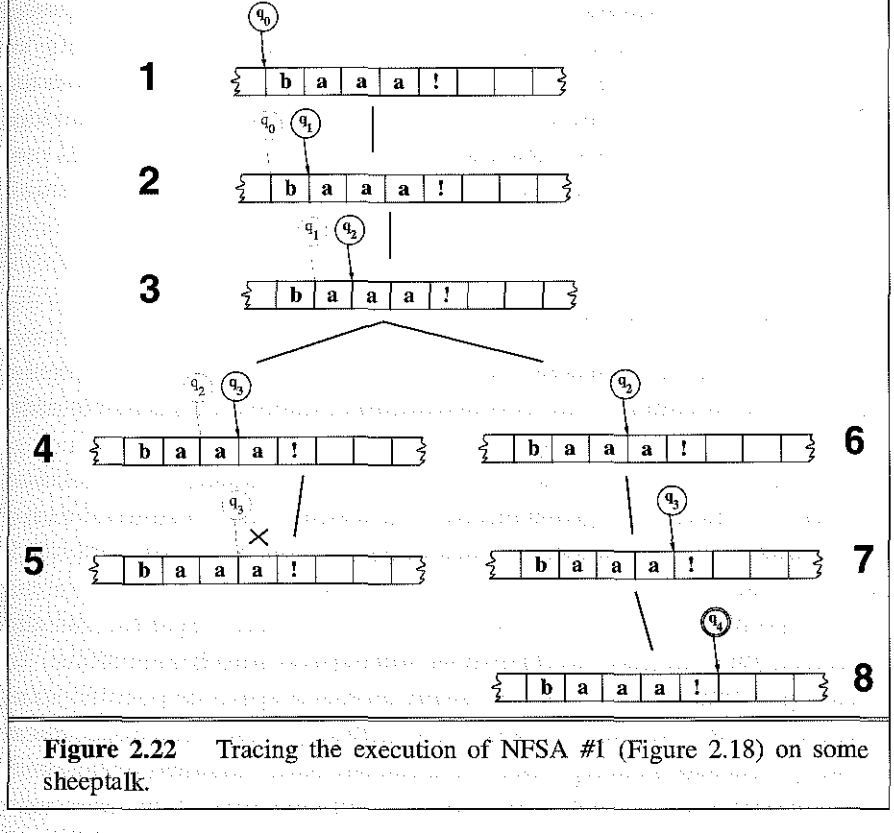


Figure 2.22 Tracing the execution of NFSA #1 (Figure 2.18) on some sheeptalk.

Figure 2.22 illustrates the progress of ND-RECOGNIZE as it attempts to handle the input `baaa!`. Each strip illustrates the state of the algorithm at a given point in its processing. The *current-search-state* variable is captured by the solid bubbles representing the machine-state along with the arrow representing progress on the tape. Each strip lower down in the figure represents progress from one *current-search-state* to the next.

Little of interest happens until the algorithm finds itself in state q_2 while looking at the second a on the tape. An examination of the entry for transition-table[q_2,a] returns both q_2 and q_3. Search states are created for each of these choices and placed on the agenda. Unfortunately, our algorithm chooses to move to state q_3, a move that results in neither an accept state nor any new states since the entry for transition-table[q_3, a] is empty. At this point, the algorithm simply asks the agenda for a new state to pursue. Since the choice of returning to q_2 from q_2 is the only unexamined choice on the agenda it is returned with the tape pointer advanced to the next a. Some-

what diabolically, ND-RECOGNIZE finds itself faced with the same choice. The entry for transition-table[q_2,a] still indicates that looping back to q_2 or advancing to q_3 are valid choices. As before, states representing both are placed on the agenda. These search states are not the same as the previous ones since their tape index values have advanced. This time the agenda provides the move to q_3 as the next move. The move to q_4, and success, is then uniquely determined by the tape and the transition-table.

Recognition as Search

ND-RECOGNIZE accomplishes the task of recognizing strings in a regular language by providing a way to systematically explore all the possible paths through a machine. If this exploration yields a path ending in an accept state, it accepts the string, otherwise it rejects it. This systematic exploration is made possible by the agenda mechanism, which on each iteration selects a partial path to explore and keeps track of any remaining, as yet unexplored, partial paths.

STATE-SPACE SEARCH

Algorithms such as ND-RECOGNIZE, which operate by systematically searching for solutions, are known as **state-space search** algorithms. In such algorithms, the problem definition creates a space of possible solutions; the goal is to explore this space, returning an answer when one is found or rejecting the input when the space has been exhaustively explored. In ND-RECOGNIZE, search states consist of pairings of machine-states with positions on the input tape. The state-space consists of all the pairings of machine-state and tape positions that are possible given the machine in question. The goal of the search is to navigate through this space from one state to another looking for a pairing of an accept state with an end of tape position.

The key to the effectiveness of such programs is often the *order* in which the states in the space are considered. A poor ordering of states may lead to the examination of a large number of unfruitful states before a successful solution is discovered. Unfortunately, it is typically not possible to tell a good choice from a bad one, and often the best we can do is to insure that each possible solution is eventually considered.

Careful readers may have noticed that the ordering of states in ND-RECOGNIZE has been left unspecified. We know only that unexplored states are added to the agenda as they are created and that the (undefined) function NEXT returns an unexplored state from the agenda when asked. How should the function NEXT be defined? Consider an ordering strategy where the states that are considered next are the most recently created ones. Such

a policy can be implemented by placing newly created states at the front of the agenda and having NEXT return the state at the front of the agenda when called. Thus the agenda is implemented by a **stack**. This is commonly referred to as a **depth-first search** or **Last In First Out (LIFO)** strategy. DEPTH-FIRST

Such a strategy dives into the search space following newly developed leads as they are generated. It will only return to consider earlier options when progress along a current lead has been blocked. The trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Figure 2.22 illustrates a depth-first search. The algorithm hits the first choice point after seeing `ba` when it has to decide whether to stay in q_2 or advance to state q_3. At this point, it chooses one alternative and follows it until it is sure it's wrong. The algorithm then backs up and tries another older alternative.

Depth first strategies have one major pitfall: under certain circumstances they can enter an infinite loop. This is possible either if the search space happens to be set up in such a way that a search-state can be accidentally re-visited, or if there are an infinite number of search states. We will revisit this question when we turn to more complicated search problems in parsing in Chapter 10.

The second way to order the states in the search space is to consider states in the order in which they are created. Such a policy can be implemented by placing newly created states at the back of the agenda and still have NEXT return the state at the front of the agenda. Thus the agenda is implemented via a **queue**. This is commonly referred to as a **breadth-first search** or **First In First Out (FIFO)** strategy. Consider a different trace of the execution of ND-RECOGNIZE on the string `baaa!` as shown in Figure 2.23. Again, the algorithm hits its first choice point after seeing `ba` when it had to decide whether to stay in q_2 or advance to state q_3. But now rather than picking one choice and following it up, we imagine examining all possible choices, expanding one ply of the search tree at a time. BREADTH-FIRST

Like depth-first search, breadth-first search has its pitfalls. As with depth-first if the state-space is infinite, the search may never terminate. More importantly, due to growth in the size of the agenda if the state-space is even moderately large, the search may require an impractically large amount of memory. For small problems, either depth-first or breadth-first search strategies may be adequate, although depth-first is normally preferred for its more efficient use of memory. For larger problems, more complex search techniques such as **dynamic programming** or **A*** must be used, as we will see in Chapters 7 and 10.

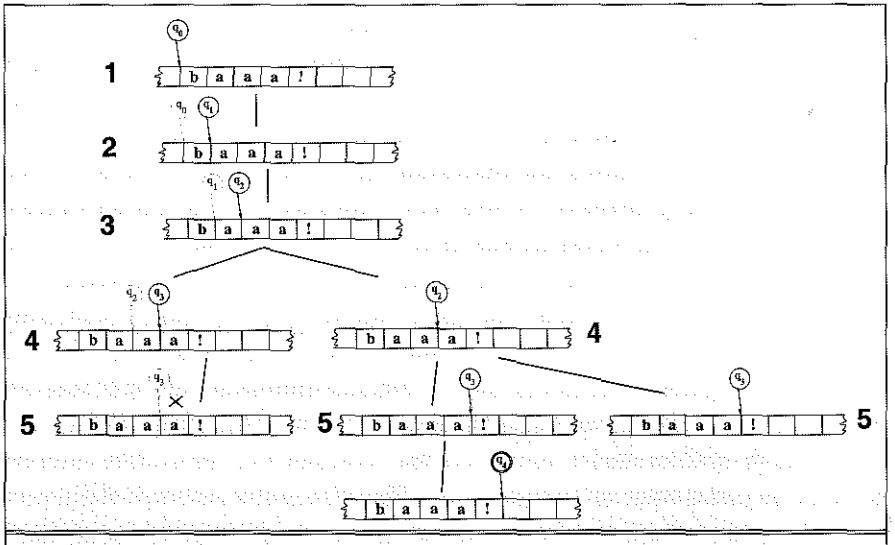


Figure 2.23 A breadth-first trace of FSA #1 on some sheeptalk.

Relating Deterministic and Non-Deterministic Automata

It may seem that allowing NFSAs to have non-deterministic features like ε-transitions would make them more powerful than DFSAs. In fact this is not the case; for any NFSA, there is an exactly equivalent DFSA. In fact there is a simple algorithm for converting an NFSA to an equivalent DFSA, although the number of states in this equivalent deterministic automaton may be much larger. See Lewis and Papadimitriou (1981) or Hopcroft and Ullman (1979) for the proof of the correspondence. The basic intuition of the proof is worth mentioning, however, and builds on the way NFSAs parse their input. Recall that the difference between NFSAs and DFSAs is that in an NFSA a state q_i may have more than one possible next state given an input i (for example q_a and q_b). The algorithm in Figure 2.21 dealt with this problem by choosing either q_a or q_b and then *backtracking* if the choice turned out to be wrong. We mentioned that a parallel version of the algorithm would follow both paths (toward q_a and q_b) simultaneously.

The algorithm for converting a NFSA to a DFSA is like this parallel algorithm; we build an automaton that has a deterministic path for every path our parallel recognizer might have followed in the search space. We imagine following both paths simultaneously, and group together into an equivalence class all the states we reach on the same input symbol (i.e., q_a and q_b). We now give a new state label to this new equivalence class state (for example

q_{ab}). We continue doing this for every possible input for every possible group of states. The resulting DFSA can have as many states as there are distinct sets of states in the original NFSA. The number of different subsets of a set with N elements is 2^N, hence the new DFSA can have as many as 2^N states.

2.3 Regular Languages and FSAs

As we suggested above, the class of languages that are definable by regular expressions is exactly the same as the class of languages that are characterizable by finite-state automata (whether deterministic or non-deterministic). Because of this, we call these languages the **regular languages**. In order to give a formal definition of the class of regular languages, we need to refer back to two earlier concepts: the alphabet Σ, which is the set of all symbols in the language, and the *empty string* ε, which is conventionally not included in Σ. In addition, we make reference to the *empty set* $\emptyset$ (which is distinct from ε). The class of regular languages (or **regular sets**) over Σ is then formally defined as follows: [1]

REGULAR LANGUAGES

1. $\emptyset$ is a regular language
2. $\forall a \in \Sigma \cup \varepsilon$, $\{a\}$ is a regular language
3. If L_1 and L_2 are regular languages, then so are:
   (a) $L_1 \cdot L_2 = \{xy | x \in L_1, y \in L_2\}$, the **concatenation** of L_1 and L_2
   (b) $L_1 \cup L_2$, the **union** or **disjunction** of L_1 and L_2
   (c) L_1^*, the **Kleene closure** of L_1

All and only the sets of languages which meet the above properties are regular languages. Since the regular languages are the set of languages characterizable by regular expressions, it must be the case that all the regular expression operators introduced in this chapter (except memory) can be implemented by the three operations which define regular languages: concatenation, disjunction/union (also called "|"), and Kleene closure. For example all the counters (`*`,`+`, `{n,m}`) are just a special case of repetition plus Kleene *. All the anchors can be thought of as individual special symbols. The square braces `[]` are a kind of disjunction (i.e., `[ab]` means "*a* or *b*", or the disjunction of *a* and *b*). Thus it is true that any regular expression can be turned into a (perhaps larger) expression which only makes use of the three primitive operations.

[1] Following van Santen and Sproat (1998), Kaplan and Kay (1994), and Lewis and Papadimitriou (1981).

Regular languages are also closed under the following operations (Σ^* means the infinite set of all possible strings formed from the alphabet Σ):

- **intersection**: if L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$, the language consisting of the set of strings that are in both L_1 and L_2.
- **difference**: if L_1 and L_2 are regular languages, then so is $L_1 - L_2$, the language consisting of the set of strings that are in L_1 but not L_2.
- **complementation**: If L_1 is a regular language, then so is $\Sigma^* - L_1$, the set of all possible strings that aren't in L_1.
- **reversal**: If L_1 is a regular language, then so is L_1^R, the language consisting of the set of reversals of all the strings in L_1.

The proof that regular expressions are equivalent to finite-state automata can be found in Hopcroft and Ullman (1979), and has two parts: showing that an automaton can be built for each regular language, and conversely that a regular language can be built for each automaton. We won't give the proof, but we give the intuition by showing how to do the first part: take any regular expression and build an automaton from it. The intuition is inductive: for the base case we build an automaton to correspond to regular expressions of a single symbol (e.g., the expression a) by creating an initial state and an accepting final state, with an arc between them labeled a. For the inductive step, we show that each of the primitive operations of a regular expression (concatenation, union, closure) can be imitated by an automaton:

- **concatenation**: We just string two FSAs next to each other by connecting all the final states of FSA_1 to the initial state of FSA_2 by an ε-transition.

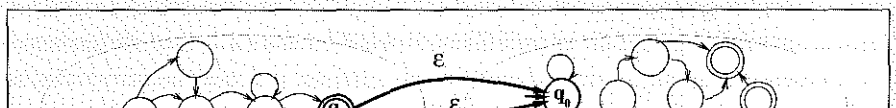


Figure 2.24 The concatenation of two FSAs.

- **closure**: We connect all the final states of the FSA back to the initial states by ε-transitions (this implements the repetition part of the Kleene *), and then put direct links between the initial and final states by ε-

transitions (this implements the possibly of having *zero* occurrences). We'd leave out this last part to implement Kleene-plus instead.

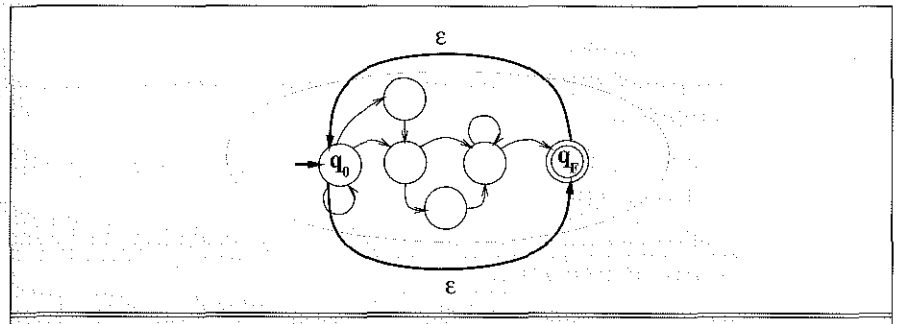


Figure 2.25 The closure (Kleene *) of an FSA.

- **union**: We add a single new initial state q'_0, and add new transitions from it to all the former initial states of the two machines to be joined.

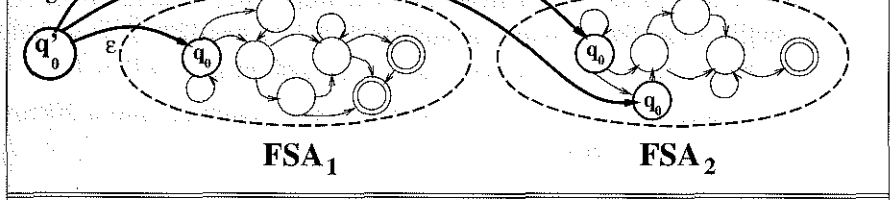


Figure 2.26 The union (|) of two FSAs.

2.4 SUMMARY

This chapter introduced the most important fundamental concept in language processing, the **finite automaton**, and the practical tool based on automaton, the **regular expression**. Here's a summary of the main points we covered about these ideas:

- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **concatenation** of symbols, **disjunction** of symbols (`[]`, `|`, and `.`), **counters** (`*`, `+`, and

{n,m}), **anchors** (^, $) and precedence operators ((,)).

- Any regular expression can be realized as a **finite state automaton** (**FSA**).
- Memory (\1 together with ()) is an advanced operation that is often considered part of regular expressions, but which cannot be realized as a finite automaton.
- An automaton implicitly defines a **formal language** as the set of strings the automaton **accepts**.
- An automaton can use any set of symbols for its vocabulary, including letters, words, or even graphic images.
- The behavior of a **deterministic** automaton (**DFSA**) is fully determined by the state it is in.
- A **non-deterministic** automaton (**NFSA**) sometimes has to make a choice between multiple paths to take given the same current state and next input.
- Any **NFSA** can be converted to a **DFSA**.
- The order in which a **NFSA** chooses the next state to explore on the agenda defines its **search strategy**. The **depth-first search** or **LIFO** strategy corresponds to the agenda-as-stack; the **breadth-first search** or **FIFO** strategy corresponds to the agenda-as-queue.
- Any regular expression can be automatically compiled into a **NFSA** and hence into a **FSA**.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Finite automata arose in the 1950s out of Turing's (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science. The Turing machine was an abstract machine with a finite control and an input/output tape. In one move, the Turing machine could read a symbol on the tape, write a different symbol on the tape, change state, and move left or right. (Thus the Turing machine differs from a finite-state automaton mainly in its ability to change the symbols on its tape).

MCCULLOCH-PITTS NEURON

Inspired by Turing's work, McCulloch and Pitts built an automata-like model of the neuron (see von Neumann, 1963, p. 319). Their model, which is now usually called the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), was a simplified model of the neuron as a kind of "computing ele-

ment" that could be described in terms of propositional logic. The model was a binary device, at any point either active or not, which took excitatory and inhibitatory input from other neurons and fired if its activation passed some fixed threshold. Based on the McCulloch-Pitts neuron, Kleene (1951) and (1956) defined the finite automaton and regular expressions, and proved their equivalence. Non-deterministic automata were introduced by Rabin and Scott (1959), who also proved them equivalent to deterministic ones.

Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His editor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the UNIX `grep` utility.

There are many general-purpose introductions to the mathematics underlying automata theory; such as Hopcroft and Ullman (1979) and Lewis and Papadimitriou (1981). These cover the mathematical foundations the simple automata of this chapter, as well as the finite-state transducers of Chapter 3, the context-free grammars of Chapter 9, and the Chomsky hierarchy of Chapter 13. Friedl (1997) is a very useful comprehensive guide to the advanced use of regular expressions.

The metaphor of problem-solving as search is basic to Artificial Intelligence (AI); more details on search can be found in any AI textbook such as Russell and Norvig (1995).

EXERCISES

2.1 Write regular expressions for the following languages: You may use either Perl notation or the minimal "algebraic" notation of Section 2.3, but make sure to say which one you are using. By "word", we mean an alphabetic string separated from other words by white space, any relevant punctuation, line breaks, and so forth.

a. the set of all alphabetic strings.

b. the set of all lowercase alphabetic strings ending in a *b*.

c. the set of all strings with two consecutive repeated words (e.g., "Humbert Humbert" and "the the" but not "the bug" or "the big bug").

d. the set of all strings from the alphabet a, b such that each a is immediately preceded and immediately followed by a b.

e. all strings which start at the beginning of the line with an integer (i.e., 1,2,3,...,10,...,10000,...) and which end at the end of the line with a word.

f. all strings which have both the word *grotto* and the word *raven* in them. (but not, for example, words like *grottos* that merely *contain* the word *grotto*).

g. write a pattern which places the first word of an English sentence in a register. Deal with punctuation.

2.2 Implement an ELIZA-like program, using substitutions such as those described on page 32. You may choose a different domain than a Rogerian psychologist, if you wish, although keep in mind that you would need a domain in which your program can legitimately do a lot of simple repeating-back.

2.3 Complete the FSA for English money expressions in Figure 2.16 as suggested in the text following the figure. You should handle amounts up to $100,000, and make sure that "cent" and "dollar" have the proper plural endings when appropriate.

2.4 Design an FSA that recognizes simple date expressions like *March 15, the 22nd of November, Christmas*. You should try to include all such "absolute" dates, (e.g. not "deictic" ones relative to the current day like *the day before yesterday*). Each edge of the graph should have a word or a set of words on it. You should use some sort of shorthand for classes of words to avoid drawing too many arcs (e.g., furniture $\rightarrow$ desk, chair, table).

2.5 Now extend your date FSA to handle deictic expressions like *yesterday, tomorrow, a week from tomorrow, the day before yesterday, Sunday, next Monday, three weeks from Saturday*.

2.6 Write an FSA for time-of-day expressions like *eleven o'clock, twelve-thirty, midnight*, or *a quarter to ten* and others.

2.7 (Due to Pauline Welby; this problem probably requires the ability to knit.) Write a regular expression (or draw an FSA) which matches all knitting patterns for scarves with the following specification: *32 stitches wide, K1P1 ribbing on both ends, stockinette stitch body, exactly two raised stripes.* All knitting patterns must include a cast-on row (to put the correct number of

stitches on the needle) and a bind-off row (to end the pattern and prevent unraveling). Here's a sample pattern for one possible scarf matching the above description:[2]

1.	Cast on 32 stitches.	*cast on; puts stitches on needle*
2.	K1 P1 across row (i.e. do (K1 P1) 16 times).	*K1P1 ribbing*
3.	Repeat instruction 2 seven more times.	*adds length*
4.	K32, P32.	*stockinette stitch*
5.	Repeat instruction 4 an additional 13 times.	*adds length*
6.	P32, P32.	*raised stripe stitch*
7.	K32, P32.	*stockinette stitch*
8.	Repeat instruction 7 an additional 251 times.	*adds length*
9.	P32, P32.	*raised stripe stitch*
10.	K32, P32.	*stockinette stitch*
11.	Repeat instruction 10 an additional 13 times.	*adds length*
12.	K1 P1 across row.	*K1P1 ribbing*
13.	Repeat instruction 12 an additional 7 times.	*adds length*
14.	Bind off 32 stitches.	*binds off row: ends pattern*

2.8 Write a regular expression for the language accepted by the NFSA in Figure 2.27.

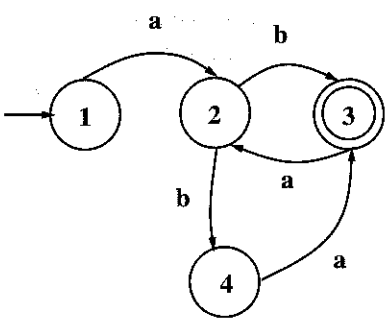


Figure 2.27 A mystery language

2.9 Currently the function D-RECOGNIZE in Figure 2.13 only solves a subpart of the important problem of finding a string in some text. Extend the algorithm to solve the following two deficiencies: (1) D-RECOGNIZE currently assumes that it is already pointing at the string to be checked, and (2)

[2] *Knit* and *purl* are two different types of stitches. The notation K*n* means do *n* knit stitches. Similarly for purl stitches. Ribbing has a striped texture—most sweaters have ribbing at the sleeves, bottom, and neck. Stockinette stitch is a series of knit and purl rows that produces a plain pattern— socks or stockings are knit with this basic pattern, hence the name.

D-RECOGNIZE fails if the string it is pointing includes as a proper substring a legal string for the FSA. That is, D-RECOGNIZE fails if there is an extra character at the end of the string.

2.10 Give an algorithm for negating a deterministic FSA. The negation of an FSA accepts exactly the set of strings that the original FSA rejects (over the same alphabet), and rejects all the strings that the original FSA accepts.

2.11 Why doesn't your previous algorithm work with NFSAs? Now extend your algorithm to negate an NFSA.

3 MORPHOLOGY AND FINITE-STATE TRANSDUCERS

A writer is someone who writes, and a stinger is something that stings. But fingers don't fing, grocers don't groce, haberdashers don't haberdash, hammers don't ham, and humdingers don't humding.

Richard Lederer, *Crazy English*

Chapter 2 introduced the regular expression, showing for example how a single search string could help a web search engine find both *woodchuck* and *woodchucks*. Hunting for singular or plural woodchucks was easy; the plural just tacks an *s* on to the end. But suppose we were looking for another fascinating woodland creatures; let's say a *fox*, and a *fish*, that surly *peccary* and perhaps a Canadian *wild goose*. Hunting for the plurals of these animals takes more than just tacking on an *s*. The plural of *fox* is *foxes*; of *peccary*, *peccaries*; and of *goose*, *geese*. To confuse matters further, fish don't usually change their form when they are plural (as Dr. Seuss points out: *one fish two fish, red fish, blue fish*).

It takes two kinds of knowledge to correctly search for singulars and plurals of these forms. **Spelling rules** tell us that English words ending in *-y* are pluralized by changing the *-y* to *-i-* and adding an *-es*. **Morphological rules** tell us that *fish* has a null plural, and that the plural of *goose* is formed by changing the vowel.

The problem of recognizing that *foxes* breaks down into the two morphemes *fox* and *-es* is called **morphological parsing**.

Key Concept #2. Parsing means taking an input and producing some sort of structure for it. PARSING

We will use the term parsing very broadly throughout this book, including many kinds of structures that might be produced; morphological, syntactic,

semantic, pragmatic; in the form of a string, or a tree, or a network. In the information retrieval domain, the similar (but not identical) problem of
STEMMING mapping from *foxes* to *fox* is called **stemming**. Morphological parsing or stemming applies to many affixes other than plurals; for example we might need to take any English verb form ending in *-ing* (*going*, *talking*, *congratulating*) and parse it into its verbal stem plus the *-ing* morpheme. So given
SURFACE the **surface** or **input form** *going*, we might want to produce the parsed form `VERB-go + GERUND-ing`. This chapter will survey the kinds of morphological knowledge that needs to be represented in different languages and introduce the main component of an important algorithm for morphological parsing: the **finite-state transducer**.

Why don't we just list all the plural forms of English nouns, and all the *-ing* forms of English verbs in the dictionary? The major reason is that *-ing*
PRODUCTIVE is a **productive** suffix; by this we mean that it applies to every verb. Similarly *-s* applies to almost every noun. So the idea of listing every noun and verb can be quite inefficient. Furthermore, productive suffixes even apply to new words (so the new word *fax* automatically can be used in the *-ing* form: *faxing*). Since new words (particularly acronyms and proper nouns) are created every day, the class of nouns in English increases constantly, and we need to be able to add the plural morpheme *-s* to each of these. Additionally, the plural form of these new nouns depends on the spelling/pronunciation of the singular form; for example if the noun ends in *-z* then the plural form is *-es* rather than *-s*. We'll need to encode these rules somewhere. Finally, we certainly cannot list all the morphological variants of every word in morphologically complex languages like Turkish, which has words like the following:

(3.1) uygarlaştıramadıklarımızdanmışsınızcasına

uygar	*+laş*	*+tır*	*+ama*	*+dık*	*+lar*	*+ımız*
civilized	+BEC	+CAUS	+NEGABLE	+PPART	+PL	+P1PL
+dan	*+mış*	*+sınız*	*+casına*			
+ABL	+PAST	+2PL	+AsIf			

"(behaving) as if you are among those whom we could not civilize/cause to become civilized"

The various pieces of this word (the **morphemes**) have these meanings:

+BEC	is "become" in English
+CAUS	is the causative voice marker on a verb
+NEGABLE	is "not able" in English

+PPart marks a past participle form
+P1PL is 1st person pl possessive agreement
+2PL is 2nd person pl
+ABL is the ablative (from/among) case marker
+AsIf is a derivational marker that forms an adverb from a finite verb form

In such languages we clearly need to parse the input since it is impossible to store every possible word. Kemal Oflazer (personal communication), who came up with this example, notes that verbs in Turkish have 40,000 forms not counting derivational suffixes; adding derivational suffixes allows a theoretically infinite number of words. This is true because, for example, any verb can be "causativized" like the example above, and multiple instances of causativization can be embedded in a single word (*You cause X to cause Y to ... do W*). Not all Turkish words look like this; Oflazer finds that the average Turkish word has about three morphemes (a root plus two suffixes). Even so, the fact that such words are possible means that it will be difficult to store all possible Turkish words in advance.

Morphological parsing is necessary for more than just information retrieval. We will need it in machine translation to realize that the French words *va* and *aller* should both translate to forms of the English verb *go*. We will also need it in spell checking; as we will see, it is morphological knowledge that will tell us that *misclam* and *antiundoggingly* are not words.

The next sections will summarize morphological facts about English and then introduce the **finite-state transducer**.

3.1 Survey of (Mostly) English Morphology

Morphology is the study of the way words are built up from smaller meaning-bearing units, **morphemes**. A morpheme is often defined as the minimal meaning-bearing unit in a language. So for example the word *fox* consists of a single morpheme (the morpheme *fox*) while the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*. MORPHEMES

As this example suggests, it is often useful to distinguish two broad classes of morphemes: **stems** and **affixes**. The exact details of the distinction vary from language to language, but intuitively, the stem is the "main" morpheme of the word, supplying the main meaning, while the affixes add "additional" meanings of various kinds. STEMS AFFIXES

Affixes are further divided into **prefixes**, **suffixes**, **infixes**, and **circumfixes**. Prefixes precede the stem, suffixes follow the stem, circumfixes do

both, and infixes are inserted inside the stem. For example, the word *eats* is composed of a stem *eat* and the suffix *-s*. The word *unbuckle* is composed of a stem *buckle* and the prefix *un-*. English doesn't have any good examples of circumfixes, but many other languages do. In German, for example, the past participle of some verbs formed by adding *ge-* to the beginning of the stem and *-t* to the end; so the past participle of the verb *sagen* (to say) is *gesagt* (said). Infixes, in which a morpheme is inserted in the middle of a word, occur very commonly for example in the Philipine language Tagalog. For example the affix *um*, which marks the agent of an action, is infixed to the Tagalog stem *hingi* "borrow" to produce *humingi*. There is one infix that occurs in some dialects of English in which taboo morpheme like "f**king" or "bl**dy" or others like it are inserted in the middle of other words ("Man-f**king-hattan", "abso-bl**dy-lutely"[1]) (McCawley, 1978).

Prefixes and suffixes are often called **concatenative morphology** since a word is composed of a number of morphemes concatenated together. A number of languages have extensive **non-concatenative morphology**, in which morphemes are combined in more complex ways. The Tagalog infixation example above is one example of non-concatenative morphology, since two morphemes (*hingi* and *um*) are intermingled. Another kind of non-concatenative morphology is called **templatic morphology** or **root-and-pattern** morphology. This is very common in Arabic, Hebrew, and other Semitic languages. In Hebrew, for example, a verb is constructed using two components: a root, consisting usually of three consonants (CCC) and carrying the basic meaning, and a template, which gives the ordering of consonants and vowels and specifies more semantic information about the resulting verb, such as the semantic voice (e.g., active, passive, middle). For example the Hebrew tri-consonantal root *lmd*, meaning 'learn' or 'study', can be combined with the active voice CaCaC template to produce the word *lamad*, 'he studied', or the intensive CiCeC template to produce the word *limed*, 'he taught', or the intensive passive template CuCaC to produce the word *lumad*, 'he was taught'.

A word can have more than one affix. For example, the word *rewrites* has the prefix *re-*, the stem *write*, and the suffix *-s*. The word *unbelievably* has a stem (*believe*) plus three affixes (*un-*, *-able*, and *-ly*). While English doesn't tend to stack more than four or five affixes, languages like Turkish can have words with nine or ten affixes, as we saw above. Languages

[1] Alan Jay Lerner, the lyricist of My Fair Lady, bowdlerized the latter to *abso-bloomin'lutely* in the lyric to "Wouldn't It Be Loverly?" (Lerner, 1978, p. 60).

that tend to string affixes together like Turkish does are called **agglutinative** languages.

There are two broad (and partially overlapping) classes of ways to form words from morphemes: **inflection** and **derivation**. Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntactic function like agreement. For example, English has the inflectional morpheme *-s* for marking the **plural** on nouns, and the inflectional morpheme *-ed* for marking the past tense on verbs. Derivation is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a *different* class, often with a meaning hard to predict exactly. For example the verb *computerize* can take the derivational suffix *-ation* to produce the noun *computerization*.

INFLECTION
DERIVATION

Inflectional Morphology

English has a relatively simple inflectional system; only nouns, verbs, and sometimes adjectives can be inflected, and the number of possible inflectional affixes is quite small.

English nouns have only two kinds of inflection: an affix that marks **plural** and an affix that marks **possessive**. For example, many (but not all) English nouns can either appear in the bare stem or **singular** form, or take a plural suffix. Here are examples of the regular plural suffix *-s*, the alternative spelling *-es*, and irregular plurals:

PLURAL
SINGULAR

	Regular Nouns		Irregular Nouns	
Singular	cat	thrush	mouse	ox
Plural	cats	thrushes	mice	oxen

While the regular plural is spelled *-s* after most nouns, it is spelled *-es* after words ending in *-s* (*ibis/ibises*) , *-z*, (*waltz/waltzes*) *-sh*, (*thrush/thrushes*) *-ch*, (*finch/finches*) and sometimes *-x* (*box/boxes*). Nouns ending in *-y* preceded by a consonant change the *-y* to *-i* (*butterfly/butterflies*).

The possessive suffix is realized by apostrophe + *-s* for regular singular nouns (*llama's*) and plural nouns not ending in *-s* (*children's*) and often by a lone apostrophe after regular plural nouns (*llamas'*) and some names ending in *-s* or *-z* (*Euripides' comedies*).

English verbal inflection is more complicated than nominal inflection. First, English has three kinds of verbs; **main verbs**, (*eat, sleep, impeach*), **modal verbs** (*can, will, should*), and **primary verbs** (*be, have, do*) (using

the terms of Quirk et al., 1985). In this chapter we will mostly be concerned with the main and primary verbs, because it is these that have inflectional
REGULAR endings. Of these verbs a large class are **regular**, that is to say all verbs of this class have the same endings marking the same functions. These regular verbs (e.g. *walk*, or *inspect*), have four morphological forms, as follow:

Morphological Form Classes	Regularly Inflected Verbs			
stem	walk	merge	try	map
-s form	walks	merges	tries	maps
-ing participle	walking	merging	trying	mapping
Past form or *-ed* participle	walked	merged	tried	mapped

These verbs are called regular because just by knowing the stem we can predict the other forms, by adding one of three predictable endings, and making some regular spelling changes (and as we will see in Chapter 4, regular pronunciation changes). These regular verbs and forms are significant in the morphology of English first because they cover a majority of the verbs, and second because the regular class is **productive**. As discussed earlier, a productive class is one that automatically includes any new words that enter the language. For example the recently-created verb *fax* (*My mom faxed me the note from cousin Everett*), takes the regular endings *-ed*, *-ing*, *-es*. (Note that the *-s* form is spelled *faxes* rather than *faxs*; we will discuss spelling rules below).

IRREGULAR VERBS The **irregular verbs** are those that have some more or less idiosyncratic forms of inflection. Irregular verbs in English often have five different forms, but can have as many as eight (e.g., the verb *be*) or as few as three (e.g. *cut* or *hit*). While constituting a much smaller class of verbs (Quirk et al. (1985) estimate there are only about 250 irregular verbs, not counting auxiliaries), this class includes most of the very frequent verbs of the language.[2] The table below shows some sample irregular forms. Note that an irregular
PRETERITE verb can inflect in the past form (also called the **preterite**) by changing its vowel (*eat/ate*), or its vowel and some consonants (*catch/caught*), or with no ending at all (*cut/cut*).

[2] In general, the more frequent a word form, the more likely it is to have idiosyncratic properties; this is due to a fact about language change; very frequent words preserve their form even if other words around them are changing so as to become more regular.

Morphological Form Classes	Irregularly Inflected Verbs		
stem	eat	catch	cut
-s form	eats	catches	cuts
-ing participle	eating	catching	cutting
Past form	ate	caught	cut
-ed participle	eaten	caught	cut

The way these forms are used in a sentence will be discussed in Chapters 8–12 but is worth a brief mention here. The *-s* form is used in the "habitual present" form to distinguish the third-person singular ending (*She jogs every Tuesday*) from the other choices of person and number (*I/you/we/they jog every Tuesday*). The stem form is used in the infinitive form, and also after certain other verbs (*I'd rather walk home, I want to walk home*). The *-ing* participle is used when the verb is treated as a noun; this particular kind of nominal use of a verb is called a **gerund** use: *Fishing is fine if you live near water.* The *-ed* participle is used in the **perfect** construction (*He's eaten lunch already*) or the passive construction (*The verdict was overturned yesterday.*).

GERUND

PERFECT

In addition to noting which suffixes can be attached to which stems, we need to capture the fact that a number of regular spelling changes occur at these morpheme boundaries. For example, a single consonant letter is doubled before adding the *-ing* and *-ed* suffixes (*beg/begging/begged*). If the final letter is "c", the doubling is spelled "ck" (*picnic/picnicking/picnicked*). If the base ends in a silent *-e*, it is deleted before adding *-ing* and *-ed* (*merge/merging/merged*). Just as for nouns, the *-s* ending is spelled *-es* after verb stems ending in *-s* (*toss/tosses*) , *-z*, (*waltz/waltzes*) *-sh*, (*wash/washes*) *-ch*, (*catch/catches*) and sometimes *-x* (*tax/taxes*). Also like nouns, verbs ending in *-y* preceded by a consonant change the *-y* to *-i* (*try/tries*).

The English verbal system is much simpler than for example the European Spanish system, which has as many as fifty distinct verb forms for each regular verb. Figure 3.1 shows just a few of the examples for the verb *amar*, 'to love'. Other languages can have even more forms than this Spanish example.

Derivational Morphology

While English inflection is relatively simple compared to other languages, derivation in English is quite complex. Recall that derivation is the combi-

Present Indicative	Imper.	Imperfect Indicative	Future	Preterite	Present Subjnct.	Conditional	Imperfect Subjnct.	Future Subjnct.
amo		amaba	amaré	amé	ame	amaría	amara	amare
amas	ama	amabas	amarás	amaste	ames	amarías	amaras	amares
	ames							
ama		amaba	amará	amó	ame	amaría	amara	amáreme
amamos		amábamos	amaremos	amamos	amemos	amaríamos	amáramos	amáremos
amáis	amad	amabais	amaréis	amasteis	améis	amaríais	amarais	amareis
	amáis							
aman		amaban	amarán	amaron	amen	amarían	amaran	amaren

Figure 3.1 To love in Spanish.

nation of a word stem with a grammatical morpheme, usually resulting in a word of a *different* class, often with a meaning hard to predict exactly.

NOMINALIZATION

A very common kind of derivation in English is the formation of new nouns, often from verbs or adjectives. This process is called **nominalization**. For example, the suffix *-ation* produces nouns from verbs ending often in the suffix *-ize* (*computerize* → *computerization*). Here are examples of some particularly productive English nominalizing suffixes.

Suffix	Base Verb/Adjective	Derived Noun
-ation	computerize (V)	computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer
-ness	fuzzy (A)	fuzziness

Adjectives can also be derived from nouns and verbs. Here are examples of a few suffixes deriving adjectives from nouns or verbs.

Suffix	Base Noun/Verb	Derived Adjective
-al	computation (N)	computational
-able	embrace (V)	embraceable
-less	clue (N)	clueless

Derivation in English is more complex than inflection for a number of reasons. One is that it is generally less productive; even a nominalizing suffix like *-ation*, which can be added to almost any verb ending in *-ize*, cannot be added to absolutely every verb. Thus we can't say **eatation* or **spellation* (we use an asterisk (*) to mark "non-examples" of English). Another is that there are subtle and complex meaning differences among nominaliz-

ing suffixes. For example *sincerity* has a subtle difference in meaning from *sincereness*.

3.2 FINITE-STATE MORPHOLOGICAL PARSING

Let's now proceed to the problem of parsing English morphology. Consider a simple example: parsing just the productive nominal plural (*-s*) and the verbal progressive (*-ing*). Our goal will be to take input forms like those in the first column below and produce output forms like those in the second column.

Input	Morphological Parsed Output
cats	`cat +N +PL`
cat	`cat +N +SG`
cities	`city +N +PL`
geese	`goose +N +PL`
goose	(`goose +N +SG`) or (`goose +V`)
gooses	`goose +V +3SG`
merging	`merge +V +PRES-PART`
caught	(`catch +V +PAST-PART`) or (`catch +V +PAST`)

The second column contains the stem of each word as well as assorted morphological **features**. These features specify additional information about the stem. For example the feature `+N` means that the word is a noun; `+SG` means it is singular, `+PL` that it is plural. We will discuss features in Chapter 11; for now, consider `+SG` to be a primitive unit that means "singular". Note that some of the input forms (like *caught* or *goose*) will be ambiguous between different morphological parses. FEATURES

In order to build a morphological parser, we'll need at least the following:

1. **lexicon:** the list of stems and affixes, together with basic information about them (whether a stem is a Noun stem or a Verb stem, etc.). LEXICON
2. **morphotactics:** the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. For example, the rule that the English plural morpheme follows the noun rather than preceding it. MORPHOTACTICS
3. **orthographic rules:** these **spelling rules** are used to model the changes that occur in a word, usually when two morphemes combine (e.g., the

$y \rightarrow ie$ spelling rule discussed above that changes *city* + *-s* to *cities* rather than *citys*).

The next part of this section will discuss how to represent a simple version of the lexicon just for the sub-problem of morphological recognition, including how to use FSAs to model morphotactic knowledge. We will then introduce the finite-state transducer (FST) as a way of modeling morphological features in the lexicon, and addressing morphological parsing. Finally, we show how to use FSTs to model orthographic rules.

The Lexicon and Morphotactics

A lexicon is a repository for words. The simplest possible lexicon would consist of an explicit list of every word of the language (*every* word, i.e., including abbreviations ("AAA") and proper names ("Jane" or "Beijing") as follows:

a
AAA
AA
Aachen
aardvark
aardwolf
aba
abaca
aback
...

Since it will often be inconvenient or impossible, for the various reasons we discussed above, to list every word in the language, computational lexicons are usually structured with a list of each of the stems and affixes of the language together with a representation of the morphotactics that tells us how they can fit together. There are many ways to model morphotactics; one of the most common is the finite-state automaton. A very simple finite-state model for English nominal inflection might look like Figure 3.2.

The FSA in Figure 3.2 assumes that the lexicon includes regular nouns (**reg-noun**) that take the regular *-s* plural (e.g., *cat*, *dog*, *fox*, *aardvark*). These are the vast majority of English nouns since for now we will ignore the fact that the plural of words like *fox* have an inserted *e*: *foxes*. The lexicon also includes irregular noun forms that don't take *-s*, both singular **irreg-sg-noun** (*goose, mouse*) and plural **irreg-pl-noun** (*geese, mice*).

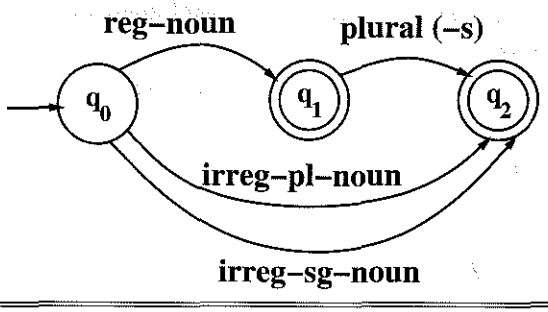


Figure 3.2 A finite-state automaton for English nominal inflection.

reg-noun	irreg-pl-noun	irreg-sg-noun	plural
fox	geese	goose	-s
cat	sheep	sheep	
dog	mice	mouse	
aardvark			

A similar model for English verbal inflection might look like Figure 3.3.

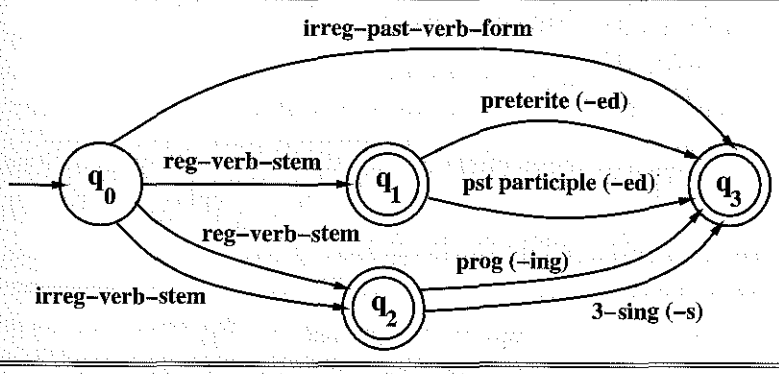


Figure 3.3 A finite-state automaton for English verbal inflection

This lexicon has three stem classes (reg-verb-stem, irreg-verb-stem, and irreg-past-verb-form), plus four more affix classes (*-ed* past, *-ed* participle, *-ing* participle, and third singular *-s*):

reg-verb-stem	irreg-verb-stem	irreg-past-verb	past	past-part	pres-part	3sg
walk	cut	caught	-ed	-ed	-ing	-s
fry	speak	ate				
talk	sing	eaten				
impeach	sang					
	spoken					

English derivational morphology is significantly more complex than English inflectional morphology, and so automata for modeling English derivation tend to be quite complex. Some models of English derivation, in fact, are based on the more complex context-free grammars of Chapter 9 (Sproat, 1993; Orgun, 1995).

As a preliminary example, though, of the kind of analysis it would require, we present a small part of the morphotactics of English adjectives, taken from Antworth (1990). Antworth offers the following data on English adjectives:

big, bigger, biggest
cool, cooler, coolest, coolly
red, redder, reddest
clear, clearer, clearest, clearly, unclear, unclearly
happy, happier, happiest, happily
unhappy, unhappier, unhappiest, unhappily
real, unreal, really

An initial hypothesis might be that adjectives can have an optional prefix (*un-*), an obligatory root (*big, cool*, etc) and an optional suffix (*-er, -est*, or *-ly*). This might suggest the the FSA in Figure 3.4.

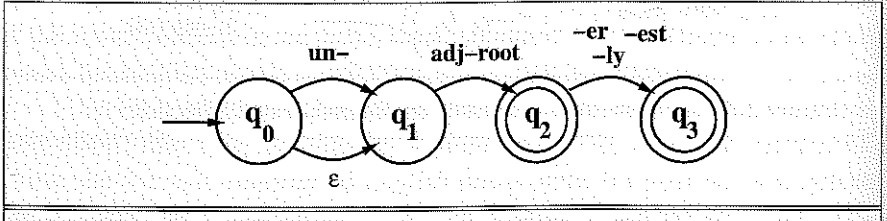


Figure 3.4 An FSA for a fragment of English adjective morphology: Antworth's Proposal #1.

Alas, while this FSA will recognize all the adjectives in the table above, it will also recognize ungrammatical forms like *unbig*, *redly*, and *realest*. We need to set up classes of roots and specify which can occur with which suffixes. So **adj-root**$_1$ would include adjectives that can occur with *un-* and *-ly* (*clear*, *happy*, and *real*) while **adj-root**$_2$ will include adjectives that can't (*big*, *cool*, and *red*). Antworth (1990) presents Figure 3.5 as a partial solution to these problems.

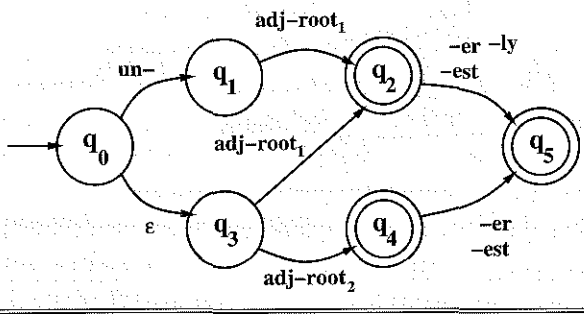


Figure 3.5 An FSA for a fragment of English adjective morphology: Antworth's Proposal #2.

This gives an idea of the complexity to be expected from English derivation. For a further example, we give in Figure 3.6 another fragment of an FSA for English nominal and verbal derivational morphology, based on Sproat (1993), Bauer (1983), and Porter (1980). This FSA models a number of derivational facts, such as the well known generalization that any verb ending in *-ize* can be followed by the nominalizing suffix *-ation* (Bauer, 1983; Sproat, 1993)). Thus since there is a word *fossilize*, we can predict the word *fossilization* by following states q_0, q_1, and q_2. Similarly, adjectives ending in *-al* or *-able* at q_5 (*equal*, *formal*, *realizable*) can take the suffix *-ity*, or sometimes the suffix *-ness* to state q_6 (*naturalness*, *casualness*). We leave it as an exercise for the reader (Exercise 3.2) to discover some of the individual exceptions to many of these constraints, and also to give examples of some of the various noun and verb classes.

We can now use these FSAs to solve the problem of **morphological recognition**; that is, of determining whether an input string of letters makes up a legitimate English word or not. We do this by taking the morphotactic FSAs, and plugging in each "sub-lexicon" into the FSA. That is, we expand each arc (e.g., the **reg-noun-stem** arc) with all the morphemes that make up the set of **reg-noun-stem**. The resulting FSA can then be defined at the level of the individual letter.

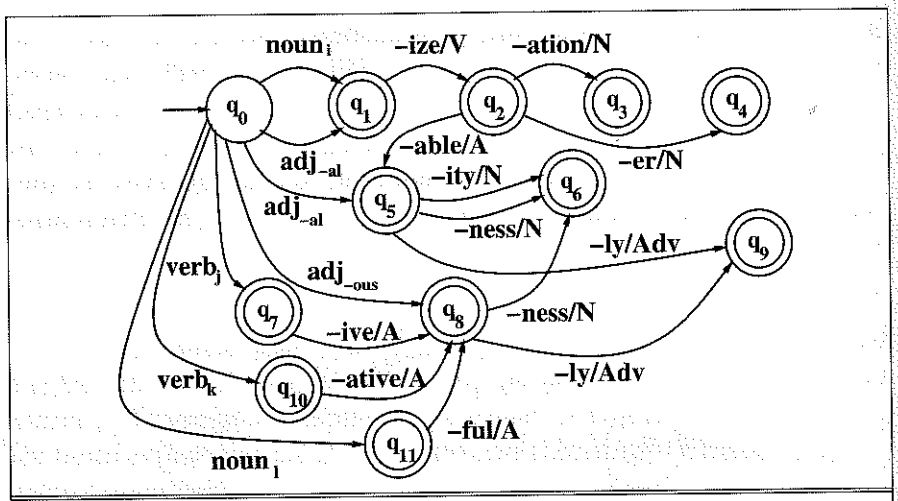


Figure 3.6 An FSA for another fragment of English derivational morphology.

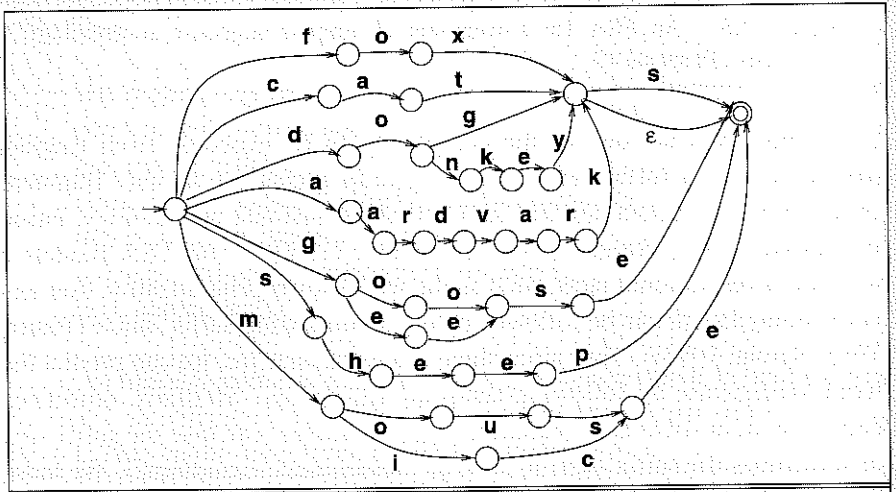


Figure 3.7 Compiled FSA for a few English nouns with their inflection. Note that this automaton will incorrectly accept the input *foxs*. We will see beginning on page 76 how to correctly deal with the inserted *e* in *foxes*.

Figure 3.7 shows the noun-recognition FSA produced by expanding the Nominal Inflection FSA of Figure 3.2 with sample regular and irregular nouns for each class. We can use Figure 3.7 to recognize strings like *aardvarks* by simply starting at the initial state, and comparing the input letter

by letter with each word on each outgoing arc, and so on, just as we saw in Chapter 2.

Morphological Parsing with Finite-State Transducers

Now that we've seen how to use FSAs to represent the lexicon and incidentally do morphological recognition, let's move on to morphological parsing. For example, given the input *cats*, we'd like to output `cat +N +PL`, telling us that cat is a plural noun. We will do this via a version of **two-level morphology**, first proposed by Koskenniemi (1983). Two-level morphology represents a word as a correspondence between a **lexical level**, which represents a simple concatenation of morphemes making up a word, and the **surface** level, which represents the actual spelling of the final word. Morphological parsing is implemented by building mapping rules that map letter sequences like *cats* on the surface level into morpheme and features sequences like `cat +N +PL` on the lexical level. Figure 3.8 shows these two levels for the word *cats*. Note that the lexical level has the stem for a word, followed by the morphological information `+N +PL` which tells us that *cats* is a plural noun.

TWO-LEVEL

SURFACE

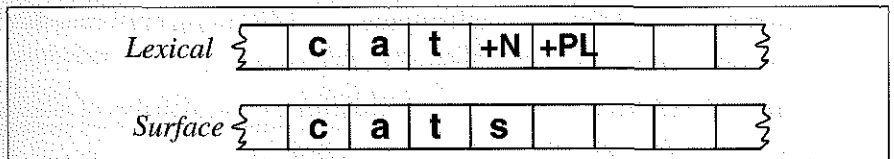


Figure 3.8 Example of the lexical and surface tapes.

The automaton that we use for performing the mapping between these two levels is the **finite-state transducer** or **FST**. A transducer maps between one set of symbols and another; a finite-state transducer does this via a finite automaton. Thus we usually visualize an FST as a two-tape automaton which recognizes or generates *pairs* of strings. The FST thus has a more general function than an FSA; where an FSA defines a formal language by defining a set of strings, an FST defines a *relation* between sets of strings. This relates to another view of an FST; as a machine that reads one string and generates another. Here's a summary of this four-fold way of thinking about transducers:

FST

- **FST as recognizer:** a transducer that takes a pair of strings as input and outputs *accept* if the string-pair is in the string-pair language, and a *reject* if it is not.
- **FST as generator:** a machine that outputs pairs of strings of the language. Thus the output is a yes or no, and a pair of output strings.
- **FST as translator:** a machine that reads a string and outputs another string
- **FST as set relater:** a machine that computes relations between sets.

MEALY MACHINE

An FST can be formally defined in a number of ways; we will rely on the following definition, based on what is called the **Mealy machine** extension to a simple FSA:

- Q: a finite set of N states $q_0, q_1, \ldots, q_N$
- Σ: a finite alphabet of complex symbols. Each complex symbol is composed of an input-output pair $i:o$; one symbol i from an input alphabet I, and one symbol o from an output alphabet O, thus $\Sigma \subseteq I \times O$. I and O may each also include the epsilon symbol ε.
- q_0: the start state
- F: the set of final states, $F \subseteq Q$
- $\delta(q, i:o)$: the transition function or transition matrix between states. Given a state $q \in Q$ and complex symbol $i:o \in \Sigma$, $\delta(q, i:o)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q.

Where an FSA accepts a language stated over a finite alphabet of single symbols, such as the alphabet of our sheep language:

$$\Sigma = \{b, a, !\} \tag{3.2}$$

an FST accepts a language stated over *pairs* of symbols, as in:

$$\Sigma = \{a:a,\ b:b,\ !:!,\ a:!,\ a:\varepsilon,\ \varepsilon:!\} \tag{3.3}$$

FEASIBLE PAIRS

In two-level morphology, the pairs of symbols in Σ are also called **feasible pairs**.

REGULAR RELATIONS

Where FSAs are isomorphic to regular languages, FSTs are isomorphic to **regular relations**. Regular relations are sets of pairs of strings, a natural extension of the regular languages, which are sets of strings. Like FSAs and regular languages, FSTs and regular relations are closed under union, although in general they are not closed under difference, complementation and intersection (although some useful subclasses of FSTs *are* closed under these operations; in general FSTs that are not augmented with the ε

are more likely to have such closure properties). Besides union, FSTs have two additional closure properties that turn out to be extremely useful:

- **inversion**: The inversion of a transducer T (T^{-1}) simply switches the input and output labels. Thus if T maps from the input alphabet I to the output alphabet O, T^{-1} maps from O to I. INVERSION
- **composition**: If T_1 is a transducer from I_1 to O_1 and T_2 a transducer from I_2 to O_2, then $T_1 \circ T_2$ maps from I_1 to O_2. COMPOSITION

Inversion is useful because it makes it easy to convert a FST-as-parser into an FST-as-generator. Composition is useful because it allows us to take two transducers that run in series and replace them with one more complex transducer. Composition works as in algebra; applying $T_1 \circ T_2$ to an input sequence S is identical to applying T_1 to S and then T_2 to the result; thus $T_1 \circ T_2(S) = T_2(T_1(S))$. We will see examples of composition below.

We mentioned that for two-level morphology it's convenient to view an FST as having two tapes. The **upper** or **lexical tape**, is composed from characters from the left side of the $a:b$ pairs; the **lower** or **surface** tape, is composed of characters from the right side of the $a:b$ pairs. Thus each symbol $a:b$ in the transducer alphabet Σ expresses how the symbol a from one tape is mapped to the symbol b on the another tape. For example $a:\varepsilon$ means that an a on the upper tape will correspond to *nothing* on the lower tape. Just as for an FSA, we can write regular expressions in the complex alphabet Σ. Since it's most common for symbols to map to themselves, in two-level morphology we call pairs like $a:a$ **default pairs**, and just refer to them by the single letter a. LEXICAL TAPE DEFAULT PAIRS

We are now ready to build an FST morphological parser out of our earlier morphotactic FSAs and lexica by adding an extra "lexical" tape and the appropriate morphological features. Figure 3.9 shows an augmentation of Figure 3.2 with the nominal morphological features (+SG and +PL) that correspond to each morpheme. Note that these features map to the empty string ε or the word/morpheme boundary symbol # since there is no segment corresponding to them on the output tape.

In order to use Figure 3.9 as a morphological noun parser, it needs to be augmented with all the individual regular and irregular noun stems, replacing the labels **regular-noun-stem** etc. In order to do this we need to update the lexicon for this transducer, so that irregular plurals like *geese* will parse into the correct stem goose +N +PL. We do this by allowing the lexicon to also have two levels. Since surface *geese* maps to underlying goose, the new lexical entry will be "g:g o:e o:e s:s e:e". Regular forms are

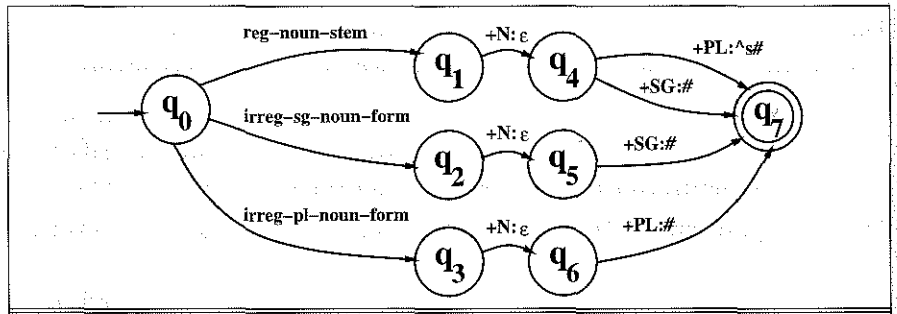


Figure 3.9 A transducer for English nominal number inflection T_{num}. Since both q_1 and q_2 are accepting states, regular nouns can have the plural suffix or not. The morpheme-boundary symbol ^ and word-boundary marker # will be discussed below.

simpler; the two-level entry for *fox* will now be "`f:f o:o x:x`", but by relying on the orthographic convention that `f` stands for `f:f` and so on, we can simply refer to it as `fox` and the form for *geese* as "`g o:e o:e s e`". Thus the lexicon will look only slightly more complex:

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat	sheep	sheep
dog	m o:i u:ε s:c e	mouse
aardvark		

Our proposed morphological parser needs to map from surface forms like *geese* to lexical forms like `goose +N +SG`. We could do this by **cascading** the lexicon above with the singular/plural automaton of Figure 3.9. Cascading two automata means running them in series with the output of the first feeding the input to the second. We would first represent the lexicon of stems in the above table as the FST T_{stems} of Figure 3.10. This FST maps e.g. *dog* to **reg-noun-stem**. In order to allow possible suffixes, T_{stems} in Figure 3.10 allows the forms to be followed by the wildcard @ **symbol**; `@:@` stands for "any feasible pair". A pair of the form `@:x`, for example will mean "any feasible pair which has `x` on the surface level", and correspondingly for the form `x:@`. The output of this FST would then feed the number automaton T_{num}.

@ SYMBOL

Instead of cascading the two transducers, we can **compose** them using the composition operator defined above. Composing is a way of taking a

cascade of transducers with many different levels of inputs and outputs and converting them into a single "two-level" transducer with one input tape and one output tape. The algorithm for composition bears some resemblance to the algorithm for determinization of FSAs from page 48; given two automata T_1 and T_2 with state sets Q_1 and Q_2 and transition functions δ_1 and δ_2, we create a new possible state (x,y) for every pair of states $x \in Q_1$ and $y \in Q_2$. Then the new automaton has the transition function:

$$\begin{aligned}\delta_3((x_a,y_a),i:o) &= (x_b,y_b) \text{ if} \\ &\exists\, c \text{ s.t. } \delta_1(x_a,i:c) = x_b \\ &\quad \text{and } \delta_2(y_a,c:o) = y_b \end{aligned} \tag{3.4}$$

The resulting composed automaton, $T_{lex} = T_{num} \circ T_{stems}$, is shown in Figure 3.11 (compare this with the FSA lexicon in Figure 3.7 on page 70).[3] Note that the final automaton still has two levels separated by the :. Because the colon was reserved for these levels, we had to use the | symbol in T_{stems} in Figure 3.10 to separate the upper and lower tapes.

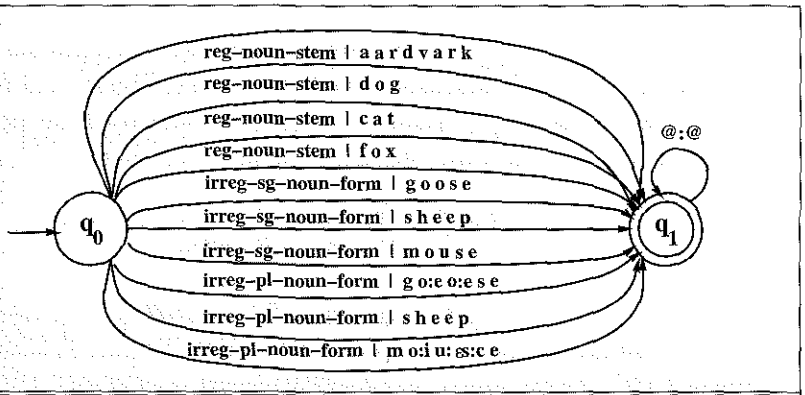


Figure 3.10 The transducer T_{stems}, which maps roots to their root-class.

This transducer will map plural nouns into the stem plus the morphological marker +PL, and singular nouns into the stem plus the morpheme +SG. Thus a surface *cats* will map to `cat +N +PL` as follows:

```
c:c a:a t:t +N:ε +PL:^s#
```

That is, c maps to itself, as do a and t, while the morphological feature +N (recall that this means "noun") maps to nothing (ε), and the feature +PL

[3] Note that for the purposes of clear exposition, Figure 3.11 has not been minimized in the way that Figure 3.7 has.

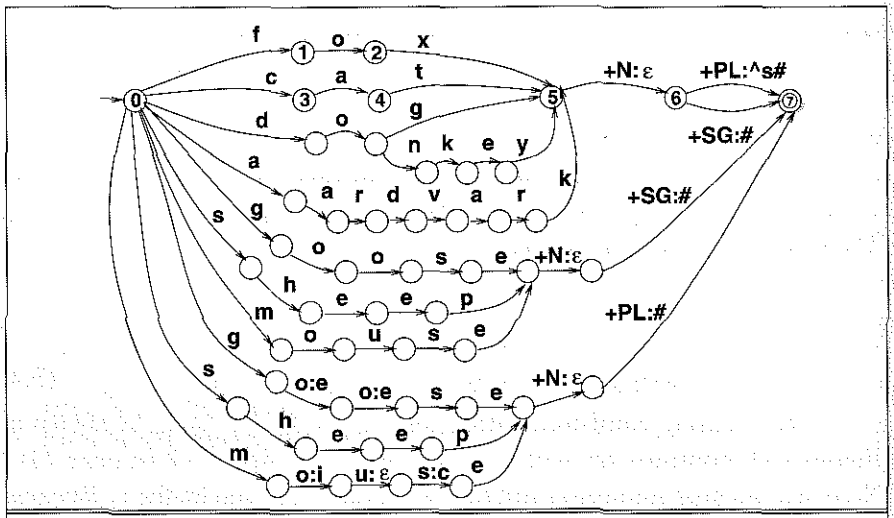


Figure 3.11 A fleshed-out English nominal inflection FST $T_{lex} = T_{num} \circ T_{stems}$.

MORPHEME BOUNDARY

WORD BOUNDARY

(meaning "plural") maps to ^s. The symbol ^ indicates a **morpheme boundary**, while the symbol # indicates a **word boundary**, Figure 3.12 refers to tapes with these morpheme boundary markers as **intermediate** tapes; the next section will show how the boundary marker is removed.

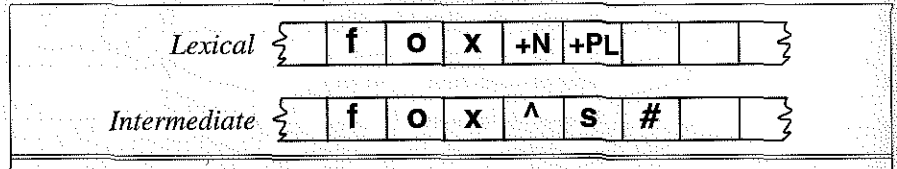


Figure 3.12 An example of the lexical and intermediate tapes.

Orthographic Rules and Finite-State Transducers

The method described in the previous section will successfully recognize words like *aardvarks* and *mice*. But just concatenating the morphemes won't work for cases where there is a spelling change; it would incorrectly reject an input like *foxes* and accept an input like *foxs*. We need to deal with the fact that English often requires spelling changes at morpheme boundaries by introducing **spelling rules** (or **orthographic rules**). This section introduces a number of notations for writing such rules and shows how to implement the rules as transducers. Some of these spelling rules:

SPELLING RULES

Name	Description of Rule	Example
Consonant doubling	1-letter consonant doubled before *-ing/-ed*	beg/begging
E deletion	Silent e dropped before *-ing* and *-ed*	make/making
E insertion	e added after *-s,-z,-x,-ch, -sh* before *-s*	watch/watches
Y replacement	*-y* changes to *-ie* before *-s*, *-i* before *-ed*	try/tries
K insertion	verbs ending with *vowel* + *-c* add *-k*	panic/panicked

We can think of these spelling changes as taking as input a simple concatenation of morphemes (the "intermediate output" of the lexical transducer in Figure 3.11) and producing as output a slightly-modified, (correctly-spelled) concatenation of morphemes. Figure 3.13 shows the three levels we are talking about: lexical, intermediate, and surface. So for example we could write an E-insertion rule that performs the mapping from the intermediate to surface levels shown in Figure 3.13. Such a rule might say some-

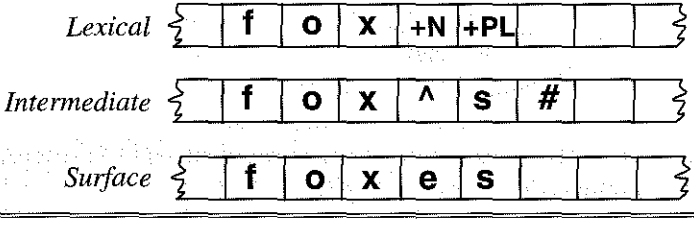


Figure 3.13 An example of the lexical, intermediate, and surface tapes. Between each pair of tapes is a two-level transducer; the lexical transducer of Figure 3.11 between the lexical and intermediate levels, and the E-insertion spelling rule between the intermediate and surface levels. The E-insertion spelling rule inserts an *e* on the surface tape when the intermediate tape has a morpheme boundary ^ followed by the morpheme *-s*.

thing like "insert an *e* on the surface tape just when the lexical tape has a morpheme ending in *x* (or *z*, etc) and the next morpheme is *-s*". Here's a formalization of the rule:

$$\varepsilon \rightarrow \text{e} / \left\{ \begin{matrix} x \\ s \\ z \end{matrix} \right\} \hat{} \_\_ \text{s\#} \tag{3.5}$$

This is the rule notation of Chomsky and Halle (1968); a rule of the form $a \rightarrow b / c \_\_ d$ means "rewrite a as b when it occurs between c and

d". Since the symbol ε means an empty transition, replacing it means inserting something. The symbol ^ indicates a morpheme boundary. These boundaries are deleted by including the symbol ^:ε in the default pairs for the transducer; thus morpheme boundary markers are deleted on the surface level by default. (Recall that the colon is used to separate symbols on the intermediate and surface forms). The # symbol is a special symbol that marks a word boundary. Thus (3.5) means "insert an *e* after a morpheme-final *x*, *s*, or *z*, and before the morpheme *s*". Figure 3.14 shows an automaton that corresponds to this rule.

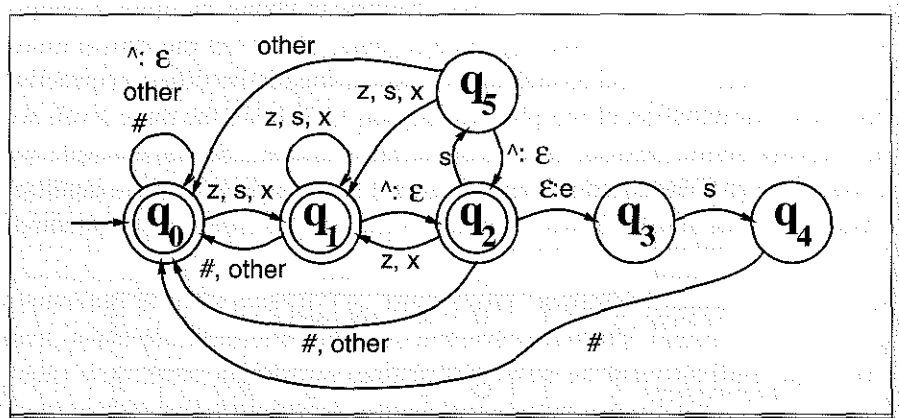


Figure 3.14 The transducer for the E-insertion rule of (3.5), extended from a similar transducer in Antworth (1990).

The idea in building a transducer for a particular rule is to express only the constraints necessary for that rule, allowing any other string of symbols to pass through unchanged. This rule is used to insure that we can only see the ε:*e* pair if we are in the proper context. So state q_0, which models having seen only default pairs unrelated to the rule, is an accepting state, as is q_1, which models having seen a *z*, *s*, or *x*. q_2 models having seen the morpheme boundary after the *z*, *s*, or *x*, and again is an accepting state. State q_3 models having just seen the E-insertion; it is not an accepting state, since the insertion is only allowed if it is followed by the *s* morpheme and then the end-of-word symbol #.

The *other* symbol is used in Figure 3.14 to safely pass through any parts of words that don't play a role in the E-insertion rule. *other* means "any feasible pair that is not in this transducer"; it is thus a version of @:@ which is context-dependent in a transducer-by-transducer way. So for example when leaving state q_0, we go to q_1 on the *z*, *s*, or *x* symbols, rather than

following the *other* arc and staying in q_0. The semantics of *other* depends on what symbols are on other arcs; since # is mentioned on some arcs, it is (by definition) not included in *other*, and thus, for example, is explicitly mentioned on the arc from q_2 to q_0.

A transducer needs to correctly reject a string that applies the rule when it shouldn't. One possible bad string would have the correct environment for the E-insertion, but have no insertion. State q_5 is used to insure that the *e* is always inserted whenever the environment is appropriate; the transducer reaches q_5 only when it has seen an *s* after an appropriate morpheme boundary. If the machine is in state q_5 and the next symbol is #, the machine rejects the string (because there is no legal transition on # from q_5). Figure 3.15 shows the transition table for the rule which makes the illegal transitions explicit with the "–" symbol.

State\Input	s:s	x:x	z:z	^:ε	ε:e	#	other
q_0:	1	1	1	0	-	0	0
q_1:	1	1	1	2	-	0	0
q_2:	5	1	1	0	3	0	0
q_3	4	-	-	-	-	-	-
q_4	-	-	-	-	-	0	-
q_5	1	1	1	2	-	-	0

Figure 3.15 The state-transition table for E-insertion rule of Figure 3.14, extended from a similar transducer in Antworth (1990).

The next section will show a trace of this E-insertion transducer running on a sample input string.

3.3 Combining FST Lexicon and Rules

We are now ready to combine our lexicon and rule transducers for parsing and generating. Figure 3.16 shows the architecture of a two-level morphology system, whether used for parsing or generating. The lexicon transducer maps between the lexical level, with its stems and morphological features, and an intermediate level that represents a simple concatenation of morphemes. Then a host of transducers, each representing a single spelling rule constraint, all run in parallel so as to map between this intermediate level and the surface level. Putting all the spelling rules in parallel is a design choice;

we could also have chosen to run all the spelling rules in series (as a long cascade), if we slightly changed each rule.

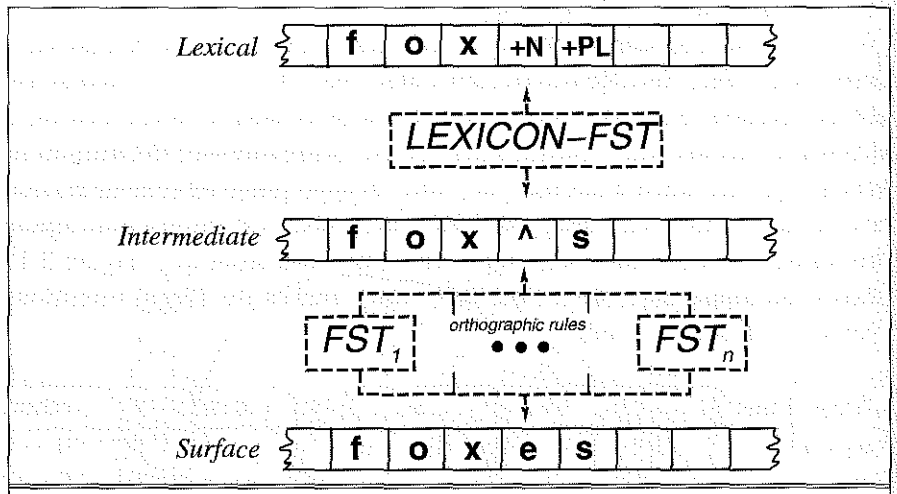


Figure 3.16 Generating or parsing with FST lexicon and rules

The architecture in Figure 3.16 is a two-level cascade of transducers. Recall that a cascade is a set of transducers in series, in which the output from one transducer acts as the input to another transducer; cascades can be of arbitrary depth, and each level might be built out of many individual transducers. The cascade in Figure 3.16 has two transducers in series: the transducer mapping from the lexical to the intermediate levels, and the collection of parallel transducers mapping from the intermediate to the surface level. The cascade can be run top-down to generate a string, or bottom-up to parse it; Figure 3.17 shows a trace of the system *accepting* the mapping from *fox^s* to *foxes*.

The power of finite-state transducers is that the exact same cascade with the same state sequences is used when the machine is generating the surface tape from the lexical tape, or when it is parsing the lexical tape from the surface tape. For example, for generation, imagine leaving the Intermediate and Surface tapes blank. Now if we run the lexicon transducer, given `fox +N +PL`, it will produce *fox^s#* on the Intermediate tape via the same states that it accepted the Lexical and Intermediate tapes in our earlier example. If we then allow all possible orthographic transducers to run in parallel, we will produce the same surface tape.

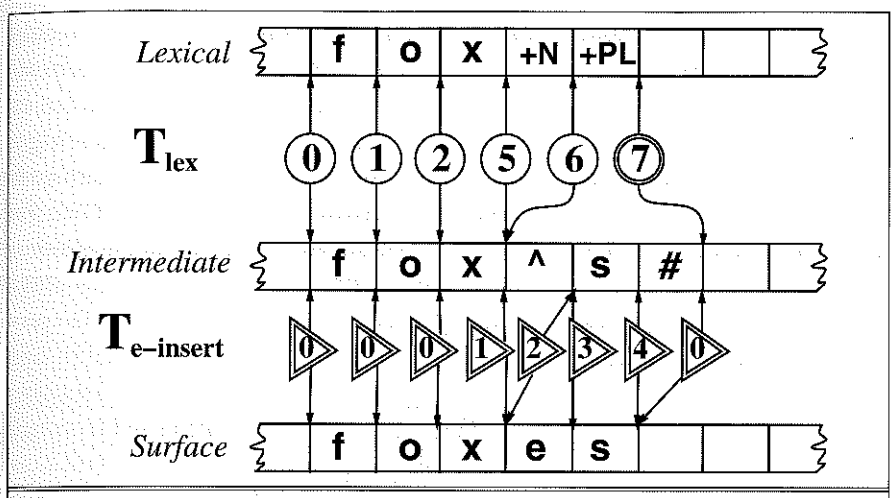


Figure 3.17 Accepting *foxes*: The lexicon transducer T_{lex} from Figure 3.11 cascaded with the E-insertion transducer in Figure 3.14.

Parsing can be slightly more complicated than generation, because of the problem of **ambiguity**. For example, *foxes* can also be a verb (albeit a rare one, meaning "to baffle or confuse"), and hence the lexical parse for *foxes* could be `fox +V +3SG` as well as `fox +N +PL`. How are we to know which one is the proper parse? In fact, for ambiguous cases of this sort, the transducer is not capable of deciding. **Disambiguating** will require some external evidence such as the surrounding words. Thus *foxes* is likely to be a noun in the sequence *I saw two foxes yesterday*, but a verb in the sequence *That trickster foxes me every time!*. We will discuss such disambiguation algorithms in Chapters 8 and 17. Barring such external evidence, the best our transducer can do is just enumerate the possible choices; so we can transduce *fox^s#* into both `fox +V +3SG` and `fox +N +PL`.

AMBIGUITY

DISAMBIGUATING

There is a kind of ambiguity that we need to handle: local ambiguity that occurs during the process of parsing. For example, imagine parsing the input verb *assess*. After seeing *ass*, our E-insertion transducer may propose that the *e* that follows is inserted by the spelling rule (for example, as far as the transducer is concerned, we might have been parsing the word *asses*). It is not until we don't see the # after *asses*, but rather run into another *s*, that we realize we have gone down an incorrect path.

Because of this non-determinism, FST-parsing algorithms need to incorporate some sort of search algorithm. Exercise 3.8 asks the reader to modify the algorithm for non-deterministic FSA recognition in Figure 2.21 in Chapter 2 to do FST parsing.

Running a cascade, particularly one with many levels, can be unwieldy. Luckily, we've already seen how to compose a cascade of transducers in series into a single more complex transducer. Transducers in parallel can be combined by **automaton intersection**. The automaton intersection algorithm just takes the Cartesian product of the states, i.e., for each state q_i in machine 1 and state q_j in machine 2, we create a new state q_{ij}. Then for any input symbol a, if machine 1 would transition to state q_n and machine 2 would transition to state q_m, we transition to state q_{nm}.

INTERSECTION

Figure 3.18 sketches how this intersection ($\wedge$) and composition ($\circ$) process might be carried out.

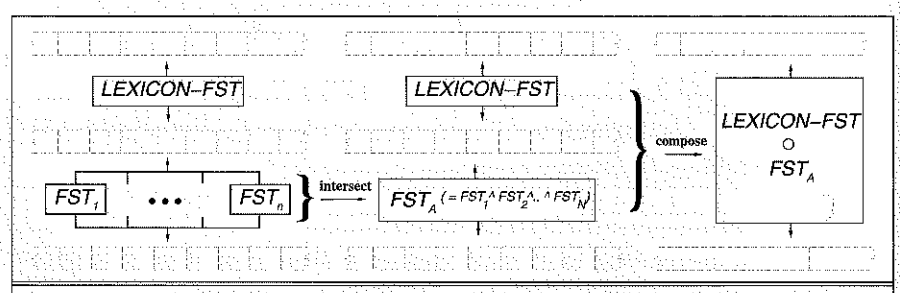


Figure 3.18 Intersection and composition of transducers.

Since there are a number of rule→FST compilers, it is almost never necessary in practice to write an FST by hand. Kaplan and Kay (1994) give the mathematics that define the mapping from rules to two-level relations, and Antworth (1990) gives details of the algorithms for rule compilation. Mohri (1997) gives algorithms for transducer minimization and determinization.

3.4 LEXICON-FREE FSTS: THE PORTER STEMMER

While building a transducer from a lexicon plus rules is the standard algorithm for morphological parsing, there are simpler algorithms that don't require the large on-line lexicon demanded by this algorithm. These are used especially in Information Retrieval (IR) tasks (Chapter 17) in which a user needs some information, and is looking for relevant documents (perhaps on the web, perhaps in a digital library database). She gives the system a query with some important characteristics of documents she desires, and the IR system retrieves what it thinks are the relevant documents. One common

type of query is Boolean combinations of relevant **keywords** or phrases, e.g. (*marsupial OR kangaroo OR koala*). The system then returns documents that have these words in them. Since a document with the word *marsupials* might not match the keyword *marsupial*, some IR systems first run a stemmer on the keywords and on the words in the document. Since morphological parsing in IR is only used to help form equivalence classes, the details of the suffixes are irrelevant; what matters is determining that two words have the same stem. KEYWORDS

One of the most widely used such **stemming** algorithms is the simple and efficient Porter (1980) algorithm, which is based on a series of simple cascaded rewrite rules. Since cascaded rewrite rules are just the sort of thing that could be easily implemented as an FST, we think of the Porter algorithm as a lexicon-free FST stemmer (this idea will be developed further in the exercises (Exercise 3.7). The algorithm contains rules like: STEMMING

(3.6) ATIONAL → ATE (e.g., relational → relate)

(3.7) ING → ε if stem contains vowel (e.g., motoring → motor)

The algorithm is presented in detail in Appendix B.

Do stemmers really improve the performance of information retrieval engines? One problem is that stemmers are not perfect. For example Krovetz (1993) summarizes the following kinds of errors of omission and of commission in the Porter algorithm:

Errors of Commission		Errors of Omission	
organization	organ	European	Europe
doing	doe	analysis	analyzes
generalization	generic	matrices	matrix
numerical	numerous	noise	noisy
policy	police	sparse	sparsity
university	universe	explain	explanation
negligible	negligent	urgency	urgent

Krovetz also gives the results of a number of experiments testing whether the Porter stemmer actually improved IR performance. Overall he found some improvement, especially with smaller documents (the larger the document, the higher the chance the keyword will occur in the exact form used in the query). Since any improvement is quite small, IR engines often don't use stemming.

3.5 HUMAN MORPHOLOGICAL PROCESSING

In this section we look at psychological studies to learn how multi-morphemic words are represented in the minds of speakers of English. For example, consider the word *walk* and its inflected forms *walks*, and *walked*. Are all three in the human lexicon? Or merely *walk* plus as well as *-ed* and *-s*? How about the word *happy* and its derived forms *happily* and *happiness*? We can imagine two ends of a theoretical spectrum of representations. The **full listing** hypothesis proposes that all words of a language are listed in the mental lexicon without any internal morphological structure. On this view, morphological structure is simply an epiphenomenon, and *walk*, *walks*, *walked*, *happy*, and *happily* are all separately listed in the lexicon. This hypothesis is certainly untenable for morphologically complex languages like Turkish (Hankamer (1989) estimates Turkish as 200 billion possible words). The **minimum redundancy** hypothesis suggests that only the constituent morphemes are represented in the lexicon, and when processing *walks*, (whether for reading, listening, or talking) we must access both morphemes (*walk* and *-s*) and combine them.

FULL LISTING

MINIMUM REDUNDANCY

Most modern experimental evidence suggests that neither of these is completely true. Rather, some kinds of morphological relationships are mentally represented (particularly inflection and certain kinds of derivation), but others are not, with those words being fully listed. Stanners et al. (1979), for example, found that derived forms (*happiness*, *happily*) are stored separately from their stem (*happy*), but that regularly inflected forms (*pouring*) are not distinct in the lexicon from their stems (*pour*). They did this by using a repetition priming experiment. In short, repetition priming takes advantage of the fact that a word is recognized faster if it has been seen before (if it is **primed**). They found that *lifting* primed *lift*, and *burned* primed *burn*, but for example *selective* didn't prime *select*. Figure 3.19 sketches one possible representation of their finding:

PRIMED

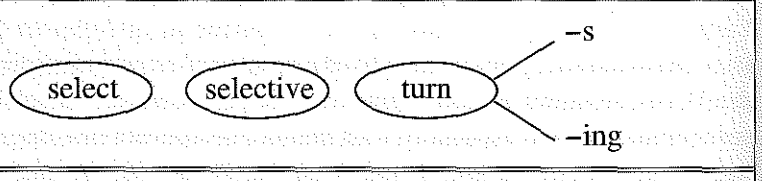


Figure 3.19 Stanners et al. (1979) result: Different representations of inflection and derivation.

In a more recent study, Marslen-Wilson et al. (1994) found that *spoken* derived words can prime their stems, but only if the meaning of the derived form is closely related to the stem. For example *government* primes *govern*, but *department* does not prime *depart*. Grainger et al. (1991) found similar results with prefixed words (but not with suffixed words). Marslen-Wilson et al. (1994) represent a model compatible with their own findings as follows:

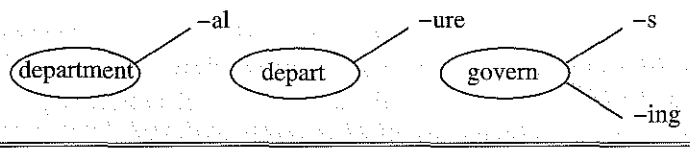


Figure 3.20 Marslen-Wilson et al. (1994) result: Derived words are linked to their stems only if semantically related

Other evidence that the human lexicon represents some morphological structure comes from **speech errors**, also called **slips of the tongue**. In normal conversation, speakers often mix up the order of the words or initial sounds:

if you break it it'll drop
I don't have time to work to watch television because I have to work

But inflectional and derivational affixes can also appear separately from their stems, as these examples from Fromkin and Ratner (1998) and Garrett (1975) show:

it's not only us who have screw looses (for "screws loose")
words of rule formation (for "rules of word formation")
easy enoughly (for "easily enough")
which by itself is the most unimplausible sentence you can imagine

The ability of these affixes to be produced separately from their stem suggests that the mental lexicon must contain some representation of the morphological structure of these words.

In summary, these results suggest that morphology does play a role in the human lexicon, especially productive morphology like inflection. They also emphasize the important of semantic generalizations across words, and suggest that the human auditory lexicon (representing words in terms of their sounds) and the orthographic lexicon (representing words in terms of letters)

may have similar structures. Finally, it seems that many properties of language processing, like morphology, may apply equally (or at least similarly) to language **comprehension** and language **production**.

3.6 SUMMARY

This chapter introduced **morphology**, the arena of language processing dealing with the subparts of words, and the **finite-state transducer**, the computational device that is commonly used to model morphology. Here's a summary of the main points we covered about these ideas:

- **Morphological parsing** is the process of finding the constituent **morphemes** in a word (e.g., `cat +N +PL` for *cats*).
- English mainly uses **prefixes** and **suffixes** to express **inflectional** and **derivational** morphology.
- English **inflectional** morphology is relatively simple and includes person and number agreement (*-s*) and tense markings (*-ed* and *-ing*).
- English **derivational** morphology is more complex and includes suffixes like *-ation*, *-ness*, *-able* as well as prefixes like *co-* and *re-*.
- Many constraints on the English **morphotactics** (allowable morpheme sequences) can be represented by finite automata.
- **Finite-state transducers** are an extension of finite-state automata that can generate output symbols.
- **Two-level morphology** is the application of finite-state transducers to morphological representation and parsing.
- **Spelling rules** can be implemented as transducers.
- There are automatic transducer-compilers that can produce a transducer for any simple rewrite rule.
- The lexicon and spelling rules can be combined by **composing** and **intersecting** various transducers.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It is not as accurate as a transducer model that includes a lexicon, but may be preferable for applications like **information retrieval** in which exact morphological structure is not needed.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Despite the close mathematical similarity of finite-state transducers to finite-state automata, the two models grew out of somewhat different traditions. Chapter 2 described how the finite automaton grew out of Turing's (1936) model of algorithmic computation, and McCulloch and Pitts finite-state-like models of the neuron. The influence of the Turing machine on the transducer was somewhat more indirect. Huffman (1954) proposed what was essentially a state-transition table to model the behavior of sequential circuits, based on the work of Shannon (1938) on an algebraic model of relay circuits. Based on Turing and Shannon's work, and unaware of Huffman's work, Moore (1956) introduced the term **finite automaton** for a machine with a finite number of states with an alphabet of input symbols and an alphabet of output symbols. Mealy (1955) extended and synthesized the work of Moore and Huffman.

The finite automata in Moore's original paper, and the extension by Mealy differed in an important way. In a Mealy machine, the input/output symbols are associated with the transitions between states. The finite-state transducers in this chapter are Mealy machines. In a Moore machine, the input/output symbols are associated with the state; we will see examples of Moore machines in Chapter 5 and Chapter 7. The two types of transducers are equivalent; any Moore machine can be converted into an equivalent Mealy machine and vice versa.

Many early programs for morphological parsing used an **affix-stripping** approach to parsing. For example Packard's (1973) parser for ancient Greek iteratively stripped prefixes and suffixes off the input word, making note of them, and then looked up the remainder in a lexicon. It returned any root that was compatible with the stripped-off affixes. This approach is equivalent to the **bottom-up** method of parsing that we will discuss in Chapter 10.

AMPLE (A Morphological Parser for Linguistic Exploration) (Weber and Mann, 1981; Weber et al., 1988; Hankamer and Black, 1991) is another early bottom-up morphological parser. It contains a lexicon with all possible surface variants of each morpheme (these are called **allomorphs**), together with constraints on their occurrence (for example in English the *-es* allomorph of the plural morpheme can only occur after s, x, z, sh, or ch). The system finds every possible sequence of morphemes which match the input and then filters out all the sequences which have failing constraints.

An alternative approach to morphological parsing is called *generate-and-test* or *analysis-by-synthesis* approach. Hankamer's (1986) keCi is a morphological parser for Turkish which is guided by a finite-state representation of Turkish morphemes. The program begins with a morpheme that might match the left edge of the word, and applies every possible phonological rule to it, checking each result against the input. If one of the outputs succeeds, the program then follows the finite-state morphotactics to the next morpheme and tries to continue matching the input.

The idea of modeling spelling rules as finite-state transducers is really based on Johnson's (1972) early idea that phonological rules (to be discussed in Chapter 4) have finite-state properties. Johnson's insight unfortunately did not attract the attention of the community, and was independently discovered by Roland Kaplan and Martin Kay, first in an unpublished talk (Kaplan and Kay, 1981) and then finally in print (Kaplan and Kay, 1994) (see page 15 for a discussion of multiple independent discoveries). Kaplan and Kay's work was followed up and most fully worked out by Koskenniemi (1983), who described finite-state morphological rules for Finnish. Karttunen (1983) built a program called KIMMO based on Koskenniemi's models. Antworth (1990) gives many details of two-level morphology and its application to English. Besides Koskenniemi's work on Finnish and that of Antworth (1990) on English, two-level or other finite-state models of morphology have been worked out for many languages, such as Turkish (Oflazer, 1993) and Arabic (Beesley, 1996). Antworth (1990) summarizes a number of issues in finite-state analysis of languages with morphologically complex processes like infixation and reduplication (e.g., Tagalog) and gemination (e.g., Hebrew). Karttunen (1993) is a good summary of the application of two-level morphology specifically to phonological rules of the sort we will discuss in Chapter 4. Barton et al. (1987) bring up some computational complexity problems with two-level models, which are responded to by Koskenniemi and Church (1988).

Students interested in further details of the fundamental mathematics of automata theory should see Hopcroft and Ullman (1979) or Lewis and Papadimitriou (1981). Mohri (1997) and Roche and Schabes (1997b) give additional algorithms and mathematical foundations for language applications, including, for example, the details of the algorithm for transducer minimization. Sproat (1993) gives a broad general introduction to computational morphology.

EXERCISES

3.1 Add some adjectives to the adjective FSA in Figure 3.5.

3.2 Give examples of each of the noun and verb classes in Figure 3.6, and find some exceptions to the rules.

3.3 Extend the transducer in Figure 3.14 to deal with `sh` and `ch`.

3.4 Write a transducer(s) for the K insertion spelling rule in English.

3.5 Write a transducer(s) for the consonant doubling spelling rule in English.

3.6 The Soundex algorithm (Odell and Russell, 1922; Knuth, 1973) is a method commonly used in libraries and older Census records for representing people's names. It has the advantage that versions of the names that are slightly misspelled or otherwise modified (common, for example, in handwritten census records) will still have the same representation as correctly-spelled names. (e.g., Jurafsky, Jarofsky, Jarovsky, and Jarovski all map to J612).

a. Keep the first letter of the name, and drop all occurrences of non-initial a, e, h, i, o, u, w, y

b. Replace the remaining letters with the following numbers:

b, f, p, v $\rightarrow$ 1
c, g, j, k, q, s, x, z $\rightarrow$ 2
d, t $\rightarrow$ 3
l $\rightarrow$ 4
m, n $\rightarrow$ 5
r $\rightarrow$ 6

c. Replace any sequences of identical numbers with a single number (i.e., 666 $\rightarrow$ 6)

d. Convert to the form `Letter Digit Digit Digit` by dropping digits past the third (if necessary) or padding with trailing zeros (if necessary).

The exercise: write a FST to implement the Soundex algorithm.

3.7 Implement one of the steps of the Porter Stemmer as a transducer.

3.8 Write the algorithm for parsing a finite-state transducer, using the pseudo-code introduced in Chapter 2. You should do this by modifying the algorithm ND-RECOGNIZE in Figure 2.21 in Chapter 2.

3.9 Write a program that takes a word and, using an on-line dictionary, computes possible anagrams of the word, each of which is a legal word.

3.10 In Figure 3.14, why is there a *z, s, x* arc from q_5 to q_1?

4 COMPUTATIONAL PHONOLOGY AND TEXT-TO-SPEECH

You like po-tay-to and I like po-tah-to.
You like to-may-to and I like to-mah-to.
Po-tay-to, po-tah-to,
To-may-to, to-mah-to,
Let's call the whole thing off!

George and Ira Gershwin, *Let's Call the Whole Thing Off* from *Shall We Dance*, 1937

The debate between the "whole language" and "phonics" methods of teaching reading to children seems at very glance like a purely modern educational debate. Like many modern debates, however, this one recapitulates an important historical dialectic, in this case in writing systems. The earliest independently-invented writing systems (Sumerian, Chinese, Mayan) were mainly logographic: one symbol represented a whole word. But from the earliest stages we can find, most such systems contain elements of syllabic or phonemic writing systems, in which symbols are used to represent the sounds that make up the words. Thus the Sumerian symbol pronounced *ba* and meaning "ration" could also function purely as the sound /ba/. Even modern Chinese, which remains primarily logographic, uses sound-based characters to spell out foreign words and especially geographical names. Purely sound-based writing systems, whether syllabic (like Japanese *hiragana* or *katakana*), alphabetic (like the Roman alphabet used in this book), or consonantal (like Semitic writing systems), can generally be traced back to these early logo-syllabic systems, often as two cultures came together. Thus the Arabic, Aramaic, Hebrew, Greek, and Roman systems all derive from a West Semitic script that is presumed to have been modified by Western Semitic mercenaries from a cursive form of Egyptian hieroglyphs. The

Japanese syllabaries were modified from a cursive form of a set of Chinese characters which were used to represent sounds. These Chinese characters themselves were used in Chinese to phonetically represent the Sanskrit in the Buddhist scriptures that were brought to China in the Tang dynasty.

Whatever its origins, the idea implicit in a sound-based writing system, that the spoken word is composed of smaller units of speech, is the Ur-theory that underlies all our modern theories of phonology. In the next four chapters we begin our exploration of these ideas, as we introduce the fundamental insights and algorithms necessary to understand modern speech recognition and speech synthesis technology, and the related branch of linguistics called **computational phonology**.

Let's begin by defining these areas. The core task of automatic speech recognition is take an acoustic waveform as input and produce as output a string of words. Conversely, the core task of text-to-speech synthesis is to take a sequence of text words and produce as output an acoustic waveform. The uses of speech recognition and synthesis are manifold, including automatic dictation/transcription, speech-based interfaces to computers and telephones, voice-based input and output for the disabled, and many others that will be discussed in greater detail in Chapter 7.

This chapter will focus on an important part of both speech recognition and text-to-speech systems: how words are pronounced in terms of individual speech units called **phones**. A speech recognition system needs to have a pronunciation for every word it can recognize, and a text-to-speech system needs to have a pronunciation for every word it can say. The first section of this chapter will introduce **phonetic alphabets** for describing pronunciation,

PHONETICS

part of the field of **phonetics**. We then introduce **articulatory phonetics**, the

ARTICULATORY PHONETICS

study of how speech sounds are produced by articulators in the mouth.

Modeling pronunciation would be much simpler if a given phone was always pronounced the same in every context. Unfortunately this is not the case. As we will see, the phone [t] is pronounced very differently in different phonetic environments. **Phonology** is the area of linguistics that describes the systematic way that sounds are differently realized in different environments, and how this system of sounds is related to the rest of the grammar. The next section of the chapter will describe the way we write **phonological rules** to describe these different realizations.

COMPUTATIONAL PHONOLOGY

We next introduce an area known as **computational phonology**. One important part of computational phonology is the study of computational mechanisms for modeling phonological rules. We will show how the spelling-rule transducers of Chapter 3 can be used to model phonology. We then

discuss computational models of **phonological learning**: how phonological rules can be automatically induced by machine learning algorithms.

Finally, we apply the transducer-based model of phonology to an important problem in text-to-speech systems: mapping from strings of letters to strings of phones. We first survey the issues involved in building a large pronunciation dictionary, and then show how the transducer-based lexicons and spelling rules of Chapter 3 can be augmented with pronunciations to map from orthography to pronunciation.

This chapter focuses on the non-probabilistic areas of computational linguistics and pronunciations modeling. Chapter 5 will turn to the role of probabilistic models, including such areas as probabilistic models of pronunciation variation and probabilistic methods for learning phonological rules.

4.1 Speech Sounds and Phonetic Transcription

The study of the pronunciation of words is part of the field of **phonetics**, the study of the speech sounds used in the languages of the world. We will be modeling the pronunciation of a word as a string of symbols which represent **phones** or **segments**. A phone is a speech sound; we will represent phones with phonetic symbols that bears some resemblance to a letter in an alphabetic language like English. So for example there is a phone represented by *l* that usually corresponds to the letter *l* and a phone represented by *p* that usually corresponds to the letter *p*. Actually, as we will see later, phones have much more variation than letters do. This chapter will only briefly touch on other aspects of phonetics such as **prosody**, which includes things like changes in pitch and duration.

PHONETICS

PHONES

This section surveys the different phones of English, particularly American English, showing how they are produced and how they are represented symbolically. We will be using two different alphabets for describing phones. The first is the **International Phonetic Alphabet** (**IPA**). The IPA is an evolving standard originally developed by the International Phonetic Association in 1888 with the goal of transcribing the sounds of all human languages. The IPA is not just an alphabet but also a set of principles for transcription, which differ according to the needs of the transcription, so the same utterance can be transcribed in different ways all according to the principles of the IPA. In the interests of brevity in this book we will focus on the symbols that are most relevant for English; thus Figure 4.1 shows a subset of the IPA symbols for transcribing consonants, while Figure 4.2 shows a subset of the IPA

IPA

IPA Symbol	ARPAbet Symbol	Word	IPA Transcription	ARPAbet Transcription
[p]	[p]	parsley	['parsli]	[p aa r s l iy]
[t]	[t]	tarragon	['tærəgɑn]	[t ae r ax g aa n]
[k]	[k]	catnip	['kætnip]	[k ae t n ix p]
[b]	[b]	bay	[beɪ]	[b ey]
[d]	[d]	dill	[dɪl]	[d ih l]
[g]	[g]	garlic	['gɑrlɨk]	[g aa r l ix k]
[m]	[m]	mint	[mɪnt]	[m ih n t]
[n]	[n]	nutmeg	['nʌtmɛg]	[n ah t m eh g
[ŋ]	[ng]	ginseng	['dʒɪnsɨŋ]	[jh ih n s ix ng]
[f]	[f]	fennel	['fɛnl̩]	[f eh n el]
[v]	[v]	clove	[klouv]	[k l ow v]
[θ]	[th]	thistle	['θɪsl̩]	[th ih s el]
[ð]	[dh]	heather	['hɛðɚ]	[h eh dh axr]
[s]	[s]	sage	[seɪdʒ]	[s ey jh]
[z]	[z]	hazelnut	['heɪzlnʌt]	[h ey z el n ah t]
[ʃ]	[sh]	squash	[skwɑʃ]	[s k w a sh]
[ʒ]	[zh]	ambrosia	[æm'broʊʒə]	[ae m b r ow zh ax]
[tʃ]	[ch]	chicory	['tʃɪkɚi]	[ch ih k axr iy]
[dʒ]	[jh]	sage	[seɪdʒ]	[s ey jh]
[l]	[l]	licorice	['lɪkɚɨʃ]	[l ih k axr ix sh]
[w]	[w]	kiwi	['kiwi]	[k iy w iy]
[r]	[r]	parsley	['pɑrsli]	[p aa r s l iy]
[j]	[y]	yew	[yu]	[y uw]
[h]	[h]	horseradish	['hɔrsrædɪʃ]	[h ao r s r ae d ih sh]
[ʔ]	[q]	uh-oh	[ʔʌʔoʊ]	[q ah q ow]
[ɾ]	[dx]	butter	['bʌɾɚ]	[b ah dx axr]
[ɾ̃]	[nx]	wintergreen	[wɪɾ̃ɚgrin]	[w ih nx axr g r i n]
[l̩]	[el]	thistle	['θɪsl̩]	[th ih s el]

Figure 4.1 IPA and ARPAbet symbols for transcription of English consonants.

symbols for transcribing vowels.[1] These tables also give the ARPAbet symbols; ARPAbet (Shoup, 1980) is another phonetic alphabet, but one that is specifically designed for American English and which uses ASCII symbols;

[1] For simplicity we use the symbol [r] for the American English "r" sound, rather than the more-standard IPA symbol [ɹ].

it can be thought of as a convenient ASCII representation of an American-English subset of the IPA. ARPAbet symbols are often used in applications where non-ASCII fonts are inconvenient, such as in on-line pronunciation dictionaries.

IPA Symbol	ARPAbet Symbol	Word	IPA Transcription	ARPAbet Transcription
[i]	[iy]	lily	['lɪli]	[l ih l iy]
[ɪ]	[ih]	lily	['lɪli]	[l ih l iy]
[eɪ]	[ey]	daisy	['deɪzi]	[d ey z i]
[ɛ]	[eh]	poinsettia	[pɔɪn'sɛɾiə]	[p oy n s eh dx iy ax]
[æ]	[ae]	aster	['æstɚ]	[ae s t axr]
[ɑ]	[aa]	poppy	['pɑpi]	[p aa p i]
[ɔ]	[ao]	orchid	['ɔrkɨd]	[ao r k ix d]
[ʊ]	[uh]	woodruff	['wʊdrʌf]	[w uh d r ah f]
[oʊ]	[ow]	lotus	['loʊɾəs]	[l ow dx ax s]
[u]	[uw]	tulip	['tulip]	[t uw l ix p]
[ʌ]	[uh]	buttercup	['bʌɾɚˌkʌp]	[b uh dx axr k uh p]
[ɝ]	[er]	bird	['bɝd]	[b er d]
[aɪ]	[ay]	iris	['aɪɹɨs]	[ay r ix s]
[aʊ]	[aw]	sunflower	['sʌnflaʊɚ]	[s ah n f l aw axr]
[ɔɪ]	[oy]	poinsettia	[pɔɪn'sɛɾiə]	[p oy n s eh dx iy ax]
[ju]	[y uw]	feverfew	[fivɚfju]	[f iy v axr f y u]
[ə]	[ax]	woodruff	['wʊdrəf]	[w uh d r ax f]
[ɨ]	[ix]	tulip	['tulip]	[t uw l ix p]
[ɚ]	[axr]	heather	['hɛðɚ]	[h eh dh axr]
[ʉ]	[ux]	dude[2]	[dʉd]	[d ux d]

Figure 4.2 IPA and ARPAbet symbols for transcription of English vowels.

Many of the IPA and ARPAbet symbols are equivalent to the Roman letters used in the orthography of English and many other languages. So for example the IPA and ARPAbet symbol [p] represents the consonant sound at

[2] The last phone, [ʉ]/[ux], is quite rare in general American English and indeed is an "extension" not present in the original ARPAbet. Labov (1994) notes that the realization of a fronted [uw] as [ux] has made it more common in (at least) Western and Northern Cities dialects of American English starting in the late 1970s. This fronting was first called to public by imitations and recordings of 'Valley Girls' speech by Moon Zappa (Zappa and Zappa, 1982). Nevertheless, for most speakers [uw] is still much more common than [ux] in words like *dude*.

the beginning of *platypus*, *puma*, and *pachyderm*, the middle of *leopard*, or the end of *antelope* (note that the final orthographic *e* of *antelope* does not correspond to any final vowel; the *p* is the last sound).

The mapping between the letters of English orthography and IPA symbols is rarely as simple as this, however. This is because the mapping between English orthography and pronunciation is quite opaque; a single letter can represent very different sounds in different contexts. Figure 4.3 shows that the English letter *c* is represented as IPA [k] in the word *cougar*, but IPA [s] in the word *civet*. Besides appearing as *c* and *k*, the sound marked as [k] in the IPA can appear as part of *x* (*fox*), as *ck* (*jackal*), and as *cc* (*raccoon*). Many other languages, for example Spanish, are much more transparent in their sound-orthography mapping than English.

Word	jackal	raccoon	cougar	civet
IPA	['dʒæ.kl]	[ræ.'kun]	['ku.gɚ]	['sɪ.vɨt]
ARPAbet	[jh ae k el]	[r ae k uw n]	[k uw g axr]	[s ih v ix t]

Figure 4.3 The mapping between IPA symbols and letters in English orthography is complicated; both IPA [k] and English orthographic [c] have many alternative realizations.

The Vocal Organs

ARTICULATORY PHONETICS

We turn now to **articulatory phonetics**, the study of how phones are produced, as the various organs in the mouth, throat, and nose modify the airflow from the lungs.

Sound is produced by the rapid movement of air. Most sounds in human spoken languages are produced by expelling air from the lungs through the windpipe (technically the **trachea**) and then out the mouth or nose. As it passes through the trachea, the air passes through the **larynx**, commonly known as the Adam's apple or voicebox. The larynx contains two small folds of muscle, the **vocal folds** (often referred to non-technically as the **vocal cords**) which can be moved together or apart. The space between these two folds is called the **glottis**. If the folds are close together (but not tightly closed), they will vibrate as air passes through them; if they are far apart, they won't vibrate. Sounds made with the vocal folds together and vibrating are called **voiced**; sounds made without this vocal cord vibration are called **unvoiced** or **voiceless**. Voiced sounds include [b], [d], [g], [v], [z], and all the English vowels, among others. Unvoiced sounds include [p], [t], [k], [f], [z], and others.

GLOTTIS

VOICED

UNVOICED

VOICELESS

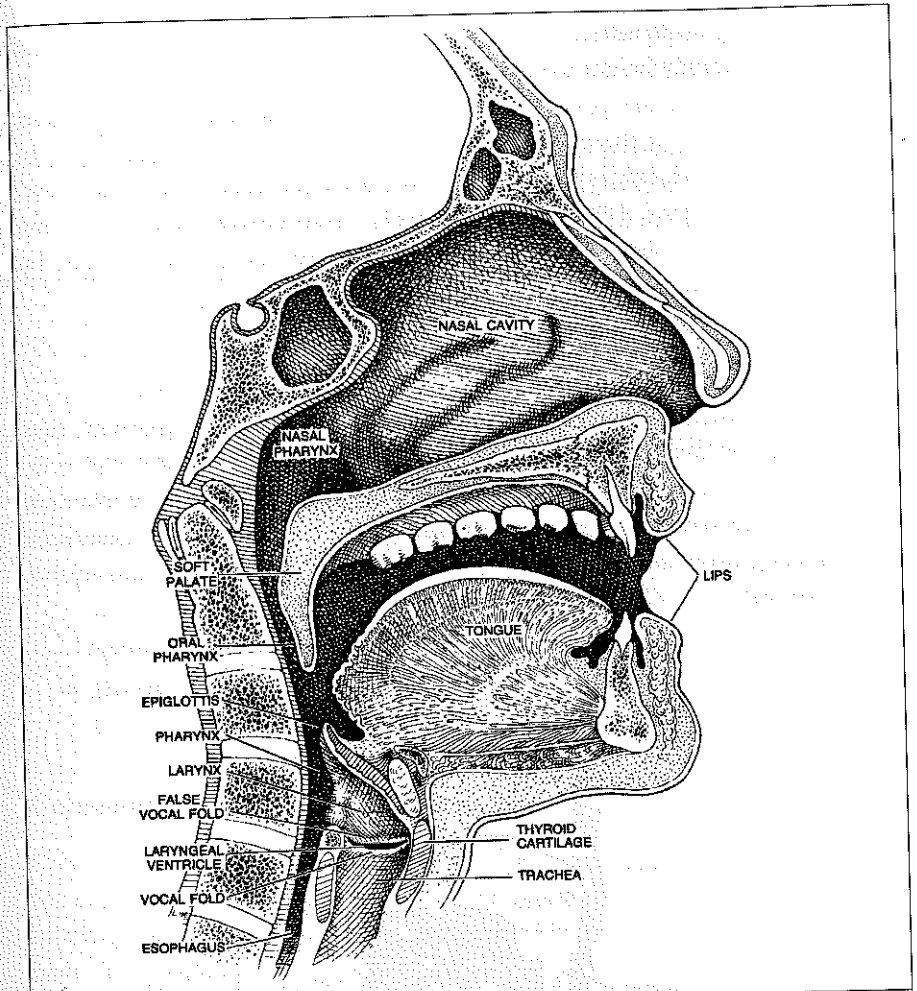


Figure 4.4 The vocal organs, shown in side view. Drawing by Laszlo Kubinyi from Sundberg (1977), ©Scientific American.

The area above the trachea is called the **vocal tract**, and consists of the **oral tract** and the **nasal tract**. After the air leaves the trachea, it can exit the body through the mouth or the nose. Most sounds are made by air passing through the mouth. Sounds made by air passing through the nose are called **nasal sounds**; nasal sounds use both the oral and nasal tracts as resonating cavities; English nasal sounds include *m*, and *n*, and *ng*.

NASAL SOUNDS

Phones are divided into two main classes: **consonants** and **vowels**. Both kinds of sounds are formed by the motion of air through the mouth,

CONSONANTS

VOWELS

throat or nose. Consonants are made by restricting or blocking the airflow in some way, and may be voiced or unvoiced. Vowels have less obstruction, are usually voiced, and are generally louder and longer-lasting than consonants. The technical use of these terms is much like the common usage; [p], [b], [t], [d], [k], [g], [f], [v], [s], [z], [r], [l], etc., are consonants; [aa], [ae], [aw], [ao], [ih], [aw], [ow], [uw], etc., are vowels. **Semivowels** (such as [y] and [w]) have some of the properties of both; they are voiced like vowels, but they are short and less syllabic like consonants.

Consonants: Place of Articulation

Because consonants are made by restricting the airflow in some way, consonants can be distinguished by where this restriction is made: the point of maximum restriction is called the **place of articulation** of a consonant. Places of articulation, shown in Figure 4.5, are often used in automatic speech recognition as a useful way of grouping phones together into equivalence classes:

PLACE

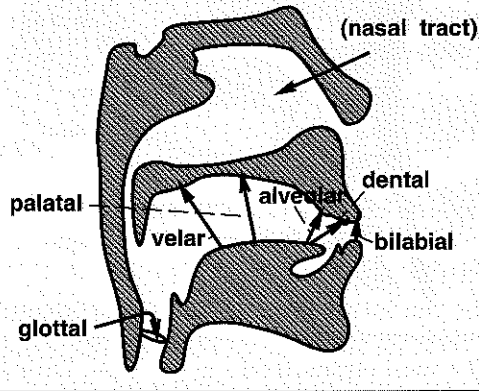


Figure 4.5 Major English places of articulation.

LABIAL

- **labial:** Consonants whose main restriction is formed by the two lips coming together have a **bilabial** place of articulation. In English these include [p] as in *possum*, [b] as in *bear*, and [m] as in *marmot*. The English **labiodental** consonants [v] and [f] are made by pressing the bottom lip against the upper row of teeth and letting the air flow through the space in the upper teeth.

DENTAL

- **dental:** Sounds that are made by placing the tongue against the teeth

are dentals. The main dentals in English are the [θ] of *thing* or the [ð] of *though*, which are made by placing the tongue behind the teeth with the tip slightly between the teeth.

- **alveolar:** The alveolar ridge is the portion of the roof of the mouth just behind the upper teeth. Most speakers of American English make the phones [s], [z], [t], and [d] by placing the tip of the tongue against the alveolar ridge. ALVEOLAR
- **palatal:** The roof of the mouth (the **palate**) rises sharply from the back of the alveolar ridge. The **palato-alveolar** sounds [ʃ] (*shrimp*), [tʃ] (*chinchilla*), [ʒ] (*Asian*), and [dʒ] (*jaguar*) are made with the blade of the tongue against this rising back of the alveolar ridge. The palatal sound [y] of *yak* is made by placing the front of the tongue up close to the palate. PALATAL PALATE
- **velar:** The **velum** or soft palate is a movable muscular flap at the very back of the roof of the mouth. The sounds [k] (*cuckoo*), [g] (*goose*), and [ŋ] (*kingfisher*) are made by pressing the back of the tongue up against the velum. VELAR VELUM
- **glottal:** The glottal stop [ʔ] is made by closing the glottis (by bringing the vocal folds together). GLOTTAL

Consonants: Manner of Articulation

Consonants are also distinguished by *how* the restriction in airflow is made, for example whether there is a complete stoppage of air, or only a partial blockage, etc. This feature is called the **manner of articulation** of a consonant. The combination of place and manner of articulation is usually sufficient to uniquely identify a consonant. Here are the major manners of articulation for English consonants: MANNER

- **stop:** A stop is a consonant in which airflow is completely blocked for a short time. This blockage is followed by an explosive sound as the air is released. The period of blockage is called the **closure** and the explosion is called the **release**. English has voiced stops like [b], [d], and [g] as well as unvoiced stops like [p], [t], and [k]. Stops are also called **plosives**. It is possible to use a more narrow (detailed) transcription style to distinctly represent the closure and release parts of a stop, both in ARPAbet and IPA-style transcriptions. For example the closure of a [p], [t], or [k] would be represented as [pcl], [tcl], or [kcl] (respectively) in the ARPAbet, and [pˀ], [tˀ], or [kˀ] (respectively) STOP

in IPA style. When this form of narrow transcription is used, the unmarked ARPABET symbols [p], [t], and [k] indicate purely the release of the consonant. We will not be using this narrow transcription style in this chapter.

NASALS
- **nasals:** The nasal sounds [n], [m], and [ŋ] are made by lowering the velum and allowing air to pass into the nasal cavity.

FRICATIVE
- **fricative:** In fricatives, airflow is constricted but not cut off completely. The turbulent airflow that results from the constriction produces a characteristic "hissing" sound. The English labiodental fricatives [f] and [v] are produced by pressing the lower lip against the upper teeth, allowing a restricted airflow between the upper teeth. The dental fricatives [θ] and [ð] allow air to flow around the tongue between the teeth. The alveolar fricatives [s] and [z] are produced with the tongue against the alveolar ridge, forcing air over the edge of the teeth. In the palato-alveolar fricatives [ʃ] and [ʒ] the tongue is at the back of the alveolar ridge forcing air through a groove formed in the tongue. The higher-pitched fricatives (in English [s], [z], [ʃ] and [ʒ]) are called **sibilants**. Stops that are followed immediately by fricatives are called **affricates**; these include English [tʃ] (*chicken*) and [dʒ] (*giraffe*).

SIBILANTS

APPROXIMANT
- **approximant:** In approximants, the two articulators are close together but not close enough to cause turbulent airflow. In English [y] (*yellow*), the tongue moves close to the roof of the mouth but not close enough to cause the turbulence that would characterize a fricative. In English [w] (*wormwood*), the back of the tongue comes close to the velum. American [ɹ] can be formed in at least two ways; with just the tip of the tongue extended and close to the palate or with the whole tongue bunched up near the palate. [l] is formed with the tip of the tongue up against the alveolar ridge or the teeth, with one or both sides of the tongue lowered to allow air to flow over it. [l] is called a **lateral** sound because of the drop in the sides of the tongue.

TAP
FLAP
- **tap:** A tap or **flap** [ɾ] is a quick motion of the tongue against the alveolar ridge. The consonant in the middle of the word *lotus* ([loʊɾəs]) is a tap in most dialects of American English; speakers of many British dialects would use a [t] instead of a tap in this word.

Vowels

Like consonants, vowels can be characterized by the position of the articulators as they are made. The two most relevant parameters for vowels are

what is called vowel **height**, which correlates roughly with the location of the highest part of the tongue, and the shape of the lips (rounded or not). Figure 4.6 shows the position of the tongue for different vowels.

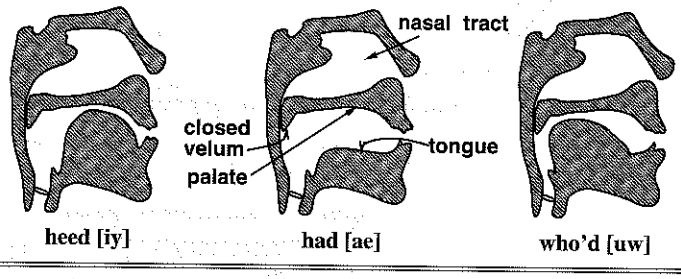


Figure 4.6 Positions of the tongue for three English vowels, high front [iy], low front [ae] and high back [uw]; tongue positions modeled after Ladefoged (1996).

In the vowel [i], for example, the highest point of the tongue is toward the front of the mouth. In the vowel [u], by contrast, the high-point of the tongue is located toward the back of the mouth. Vowels in which the tongue is raised toward the front are called **front vowels**; those in which the tongue is raised toward the back are called **back vowels**. Note that while both [ɪ] and [ɛ] are front vowels, the tongue is higher for [ɪ] than for [ɛ]. Vowels in which the highest point of the tongue is comparatively high are called **high vowels**; vowels with mid or low values of maximum tongue height are called **mid vowels** or **low vowels**, respectively.

FRONT
BACK
HIGH

Figure 4.7 shows a schematic characterization of the vowel height of different vowels. It is schematic because the abstract property **height** only correlates roughly with actual tongue positions; it is in fact a more accurate reflection of acoustic facts. Note that the chart has two kinds of vowels: those in which tongue height is represented as a point and those in which it is represented as a vector. A vowels in which the tongue position changes markedly during the production of the vowel is **diphthong**. English is particularly rich in diphthongs; many are written with two symbols in the IPA (for example the [eɪ] of *hake* or the [oʊ] of *cobra*).

DIPHTHONG

The second important articulatory dimension for vowels is the shape of the lips. Certain vowels are pronounced with the lips rounded (the same lip shape used for whistling). These **rounded** vowels include [u], [ɔ], and the diphthong [oʊ].

ROUNDED

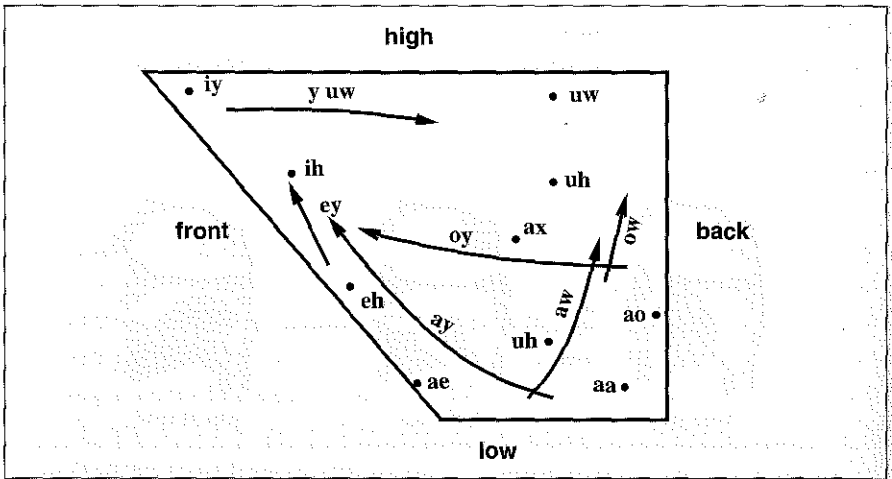


Figure 4.7 Qualities of English vowels (after Ladefoged (1993)).

Syllables

SYLLABLE

Consonants and vowels combine to make a **syllable**. There is no completely agreed-upon definition of a syllable; roughly speaking a syllable is a vowel-like sound together with some of the surrounding consonants that are most closely associated with it. The IPA period symbol [.] is used to separate syllables, so *parsley* and *catnip* have two syllables (['par.sli] and ['kæt.nip] respectively), *tarragon* has three ['tæ.rə.gan], and *dill* has one ([dɪl]). A syllable is usually described as having an optional initial consonant or set of consonants called the **onset**, followed by a vowel or vowels, followed by a final consonant or sequence of consonants called the **coda**. Thus d is the onset of [dɪl], while l is the coda. The task of breaking up a word into syllables is called **syllabification**. Although automatic syllabification algorithms exist, the problem is hard, partly because there is no agreed-upon definition of syllable boundaries. Furthermore, although it is usually clear how many syllables are in a word, Ladefoged (1993) points out there are some words (meal, teal, seal, hire, fire, hour) that can be viewed either as having one syllable or two.

ONSET

CODA

SYLLABIFICATION

In a natural sentence of American English, certain syllables are more **prominent** than others. These are called **accented** syllables. Accented syllables may be prominent because they are louder, they are longer, they are associated with a pitch movement, or any combination of the above. Since accent plays important roles in meaning, understanding exactly why a speaker

ACCENTED

chooses to accent a particular syllable is very complex. But one important factor in accent is often represented in pronunciation dictionaries. This factor is called **lexical stress**. The syllable that has lexical stress is the one that will be louder or longer if the word is accented. For example the word *parsley* is stressed in its first syllable, not its second. Thus if the word *parsley* is accented in a sentence, it is the first syllable that will be stronger. We write the symbol ['] before a syllable to indicate that it has lexical stress (e.g. ['par.sli]). This difference in lexical stress can affect the meaning of a word. For example the word *content* can be a noun or an adjective. When pronounced in isolation the two senses are pronounced differently since they have different stressed syllables (the noun is pronounced ['kɑn.tɛnt]) and the adjective [kən.'tɛnt]. Other pairs like this include *object* (noun ['ɑb.dʒɛkt] and verb [əb.'dʒɛkt]); see Cutler (1986) for more examples. Automatic disambiguation of such **homographs** is discussed in Chapter 17. The role of prosody is taken up again in Section 4.7.

LEXICAL STRESS

HOMOGRAPHS

4.2 THE PHONEME AND PHONOLOGICAL RULES

'Scuse me, while I kiss the sky
Jimi Hendrix, *Purple Haze*
'Scuse me, while I kiss this guy
Common mis-hearing of same lyrics

All [t]s are not created equally. That is, phones are often produced differently in different contexts. For example, consider the different pronunciations of [t] in the words *tunafish* and *starfish*. The [t] of *tunafish* is **aspirated**. Aspiration is a period of voicelessness after a stop closure and before the onset of voicing of the following vowel. Since the vocal cords are not vibrating, aspiration sounds like a puff of air after the [t] and before the vowel. By contrast, a [t] following an initial [s] is **unaspirated**; thus the [t] in *starfish* ([stɑrfiʃ]) has no period of voicelessness after the [t] closure. This variation in the realization of [t] is predictable: whenever a [t] begins a word or unreduced syllable in English, it is aspirated. The same variation occurs for [k]; the [k] of *sky* is often mis-heard as [g] in Jimi Hendrix's lyrics because [k] and [g] are both unaspirated. In a very detailed transcription system we could use the symbol for aspiration [^{h}]after any [t] (or [k] or [p]) which begins a word or unreduced syllable. The word *tunafish* would be transcribed [t^{h}unəfiʃ] (the ARPAbet does not have a way of marking aspiration).

UNASPIRATED

There are other contextual variants of [t]. For example, when [t] occurs between two vowels, particularly when the first is stressed, it is pronounced as a tap. Recall that a tap is a voiced sound in which the top of the tongue is curled up and back and struck quickly against the alveolar ridge. Thus the word *buttercup* is usually pronounced [bʌɾɚkʌp]/[b uh dx axr k uh p] rather than [bʌtɚkʌp]/[b uh t axr k uh p].

Another variant of [t] occurs before the dental consonant [θ]. Here the [t] becomes dentalized ([t̪]). That is, instead of the tongue forming a closure against the alveolar ridge, the tongue touches the back of the teeth.

How do we represent this relation between a [t] and its different realizations in different contexts? We generally capture this kind of pronunciation variation by positing an abstract class called the **phoneme**, which is realized as different **allophones** in different contexts. We traditionally write phonemes inside slashes. So in the above examples, /t/ is a phoneme whose allophones include [tʰ], [ɾ], and [t̪]. A phoneme is thus a kind of generalization or abstraction over different phonetic realizations. Often we equate the phonemic and the lexical levels, thinking of the lexicon as containing transcriptions expressed in terms of phonemes. When we are transcribing the pronunciations of words we can choose to represent them at this broad phonemic level; such a **broad transcription** leaves out a lot of predictable phonetic detail. We can also choose to use a **narrow transcription** that includes more detail, including allophonic variation, and uses the various diacritics. Figure 4.8 summarizes a number of allophones of /t/; Figure 4.9 shows a few of the most commonly used IPA diacritics.

PHONEME

ALLOPHONES

NARROW TRANSCRIPTION

Phone	Environment	Example	IPA
[tʰ]	in initial position	*toucan*	[tʰukʰæn]
[t]	after [s] or in reduced syllables	*starfish*	[stɑɾfɪʃ]
[ʔ]	word-finally or after vowel before [n]	*kitten*	[kʰɪʔn]
[ʔt]	sometimes word-finally	*cat*	[kʰæʔt]
[ɾ]	between vowels	*buttercup*	[bʌɾɚkʰʌp]
[t˺]	before consonants or word-finally	*fruitcake*	[frut˺kʰeɪk]
[t̪]	before dental consonants ([θ])	*eighth*	[eɪt̪θ]
[]	sometimes word-finally	*past*	[pæs]

Figure 4.8 Some allophones of /t/ in General American English.

The relationship between a phoneme and its allophones is often captured by writing a **phonological rule**. Here is the phonological rule for dentalization in the traditional notation of Chomsky and Halle (1968):

$$/t/ \rightarrow [\underset{\sqcap}{t}] \; / \_\_ \; \theta \tag{4.1}$$

In this notation, the surface allophone appears to the right of the arrow, and the phonetic environment is indicated by the symbols surrounding the underbar (__). These rules resemble the rules of two-level morphology of Chapter 3 but since they don't use multiple types of rewrite arrows, this rule is ambiguous between an obligatory or optional rule. Here is a version of the flapping rule:

$$\left/ \left\{ \begin{matrix} t \\ d \end{matrix} \right\} \right/ \rightarrow [\text{ɾ}] \; / \; \acute{\text{V}} \_\_ \; \text{V} \tag{4.2}$$

Diacritics			Suprasegmentals		
◌̥	Voiceless	[ḁ]	ˈ	Primary stress	[ˈpu.mə]
◌ʰ	Aspirated	[pʰ]	ˌ	Secondary stress	[ˈfoʊrəˌgræf]
◌̩	Syllabic	[l̩]	ː	Long	[aː]
◌̃	Nasalized	[æ̃]	ˑ	Half long	[aˑ]
◌̚	Unreleased	[t̚]	.	Syllable break	[ˈpu.mə]
◌̪	Dental	[t̪]			

Figure 4.9 Some of the IPA diacritics and symbols for suprasegmentals.

4.3 Phonological Rules and Transducers

Chapter 3 showed that spelling rules can be implemented by transducers. Phonological rules can be implemented as transducers in the same way; indeed the original work by Johnson (1972) and Kaplan and Kay (1981) on finite-state models was based on phonological rules rather than spelling rules. There are a number of different models of **computational phonology** that use finite automata in various ways to realize phonological rules. We will describe the **two-level morphology** of Koskenniemi (1983) used in Chapter 3, but the interested reader should be aware of other recent models.[3] While Chapter 3 gave examples of two-level rules, it did not talk about the

[3] One example is Bird and Ellison's (1994) model of the multi-tier representations of autosegmental phonology in which each phonological tier is represented by a finite-state automaton, and autosegmental association by the synchronization of two automata.

motivation for these rules, and the differences between traditional ordered rules and two-level rules. We will begin with this comparison.

As a first example, Figure 4.10 shows a transducer which models the application of the simplified flapping rule in (4.3):

$$/t/ \rightarrow [ɾ] \,/\, \acute{V} \_\_ V \tag{4.3}$$

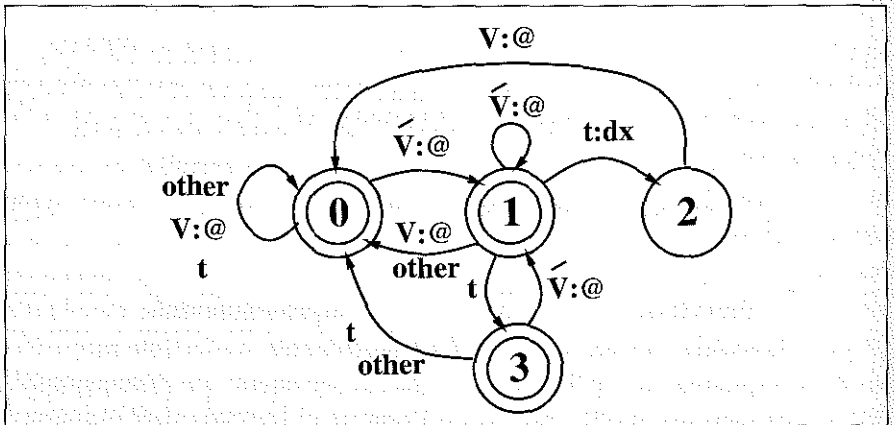


Figure 4.10 Transducer for English Flapping: ARPAbet "dx" indicates a flap, and the "other" symbol means "any feasible pair not used elsewhere in the transducer". "@" means "any symbol not used elsewhere on any arc".

The transducer in Figure 4.10 accepts any string in which flaps occur in the correct places (after a stressed vowel, before an unstressed vowel), and rejects strings in which flapping doesn't occur, or in which flapping occurs in the wrong environment. Of course the factors that flapping are actually a good deal more complicated, as we will see in Section 5.7.

In a traditional phonological system, many different phonological rules apply between the lexical form and the surface form. Sometimes these rules interact; the output from one rule affects the input to another rule. One way to implement rule-interaction in a transducer system is to run transducers in a *cascade*. Consider, for example, the rules that are needed to deal with the phonological behavior of the English noun plural suffix *-s*. This suffix is pronounced [ɨz] after the phones [s], [ʃ], [z], or [ʒ] (so *peaches* is pronounced [pitʃɨz], and *faxes* is pronounced [fæksɨz]), [z] after voiced sounds (*pigs* is pronounced [pɪgz]), and [s] after unvoiced sounds (*cats* is pronounced [kæts]). We model this variation by writing phonological rules for the realization of the morpheme in different contexts. We first need to choose one of these three forms (s, z, and ɨz) as the "lexical" pronunciation of the suffix; we

chose z only because it turns out to simplify rule writing. Next we write two phonological rules. One, similar to the E-insertion spelling rule of page 77, inserts a [ɨ] after a morpheme-final sibilant and before the plural morpheme [z]. The other makes sure that the *-s* suffix is properly realized as [s] after unvoiced consonants.

$$\varepsilon \rightarrow \text{ɨ} \;/\; [\text{+sibilant}]\ \hat{}\ \_\_\ \text{z}\ \# \tag{4.4}$$

$$\text{z} \rightarrow \text{s} \;/\; [\text{-voice}]\ \hat{}\ \_\_\ \# \tag{4.5}$$

These two rules must be *ordered*; rule (4.4) must apply before (4.5). This is because the environment of (4.4) includes z, and the rule (4.5) changes z. Consider running both rules on the lexical form *fox* concatenated with the plural *-s*:

Lexical form:	fɑksˆz
(4.4) applies:	fɑksˆɨz
(4.5) doesn't apply:	fɑksˆɨz

If the devoicing rule (4.5) was ordered first, we would get the wrong result (what would this incorrect result be?). This situation, in which one rule destroys the environment for another, is called **bleeding**:[4]

Lexical form:	fɑksˆz
(4.5) applies:	fɑksˆs
(4.4) doesn't apply:	fɑksˆs

As was suggested in Chapter 3, each of these rules can be represented by a transducer. Since the rules are ordered, the transducers would also need to be ordered. For example if they are placed in a **cascade**, the output of the first transducer would feed the input of the second transducer.

Many rules can be cascaded together this way. As Chapter 3 discussed, running a cascade, particularly one with many levels, can be unwieldy, and so transducer cascades are usually replaced with a single more complex transducer by **composing** the individual transducers.

Koskenniemi's method of **two-level morphology** that was sketchily introduced in Chapter 3 is another way to solve the problem of rule ordering. Koskenniemi (1983) observed that most phonological rules in a grammar are independent of one another; that feeding and bleeding relations between

[4] If we had chosen to represent the lexical pronunciation of *-s* as [s] rather than [z], we would have written the rule inversely to voice the *-s* after voiced sounds, but the rules would still need to be ordered; the ordering would simply flip.

rules are not the norm.[5] Since this is the case, Koskenniemi proposed that phonological rules be run in parallel rather than in series. The cases where there is rule interaction (feeding or bleeding) we deal with by slightly modifying some rules. Koskenniemi's two-level rules can be thought of as a way of expressing **declarative constraints** on the well-formedness of the lexical-surface mapping.

Two-level rules also differ from traditional phonological rules by explicitly coding when they are obligatory or optional, by using four differing **rule operators**; the ⇔ rule corresponds to traditional **obligatory** phonological rules, while the ⇒ rule implements **optional rules**:

Rule type	Interpretation
a:b ⇐ c ___ d	*a* is **always** realized as *b* in the context *c* ___ *d*
a:b ⇒ c ___ d	*a* may be realized as *b* **only** in the context *c* ___ *d*
a:b ⇔ c ___ d	*a* must be realized as *b* in context *c* ___ *d* and nowhere else
a:b /⇐ c ___ d	*a* is **never** realized as *b* in the context *c* ___ *d*

The most important intuition of the two-level rules, and the mechanism that lets them avoiding feeding and bleeding, is their ability to represent constraints on *two levels*. This is based on the use of the colon (":"), which was touched in very briefly in Chapter 3. The symbol *a:b* means a lexical *a* that maps to a surface *b*. Thus *a:b* ⇔ *:c* ___ means *a* is realized as *b* after a **surface** c. By contrast *a:b* ⇔ *c:* ___ means that *a* is realized as *b* after a **lexical** c. As discussed in Chapter 3, the symbol *c* with no colon is equivalent to *c:c* that means a lexical *c* which maps to a surface *c*.

Figure 4.11 shows an intuition for how the two-level approach avoids ordering for the ɨ-insertion and z-devoicing rules. The idea is that the z-devoicing rule maps a *lexical* z-insertion to a *surface* s and the ɨ rule refers to the *lexical* z:

The two-level rules that model this constraint are shown in (4.6) and (4.7):

$$\varepsilon : \text{ɨ} \Leftrightarrow \text{[+sibilant]: } \hat{} \text{ \_\_ z: \#} \tag{4.6}$$

$$\text{z} : \text{s} \Leftrightarrow \text{[-voice]: } \hat{} \text{ \_\_ \#} \tag{4.7}$$

As Chapter 3 discussed, there are compilation algorithms for creating automata from rules. Kaplan and Kay (1994) give the general derivation of these algorithms, and Antworth (1990) gives one that is specific to two-level rules. The automata corresponding to the two rules are shown in Figure 4.12

[5] Feeding is a situation in which one rules creates the environment for another rule and so must be run beforehand.

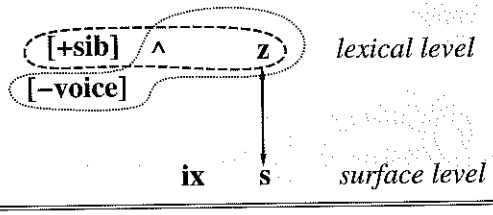


Figure 4.11 The constraints for the ɨ-insertion and z-devoicing rules both refer to a *lexical* z, not a *surface* s.

and Figure 4.13. Figure 4.12 is based on Figure 3.14 of Chapter 3; see page 78 for a reminder of how this automaton works. Note in Figure 4.12 that the plural morpheme is represented by z:, indicating that the constraint is expressed about an lexical rather than surface z.

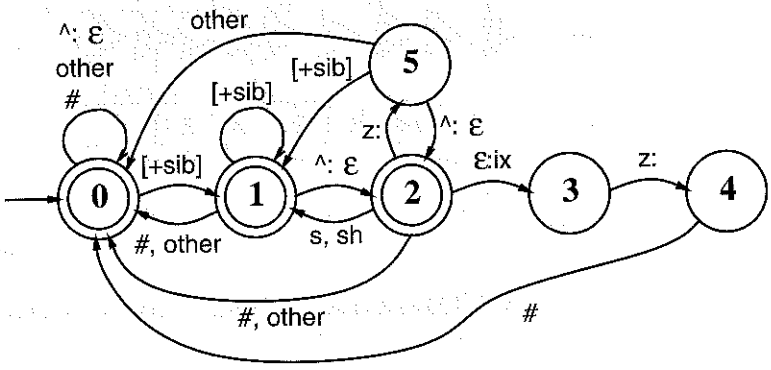


Figure 4.12 The transducer for the ɨ-insertion rule 4.4. The rule can be read *whenever a morpheme ends in a sibilant, and the following morpheme is* z, *insert* [ɨ].

Figure 4.14 shows the two automata run in parallel on the input [fɑksˆz] (the figure uses the ARPAbet notation [f aa k s ^ z]). Note that both the automata assuming the default mapping ^:ε to remove the morpheme boundary, and that both automata end in an accepting state.

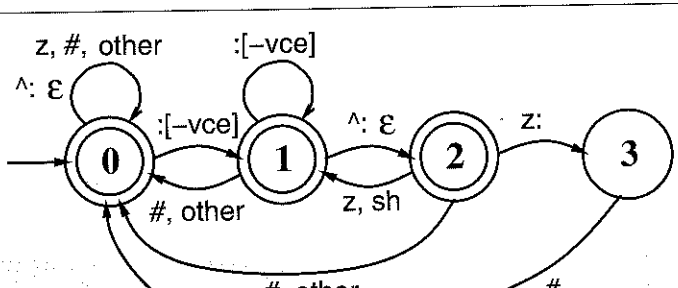


Figure 4.13 The transducer for the z-devoicing rule 4.5. This rule might be summarized *Devoice the morpheme* z *if it follows a morpheme-final voiceless consonant.*

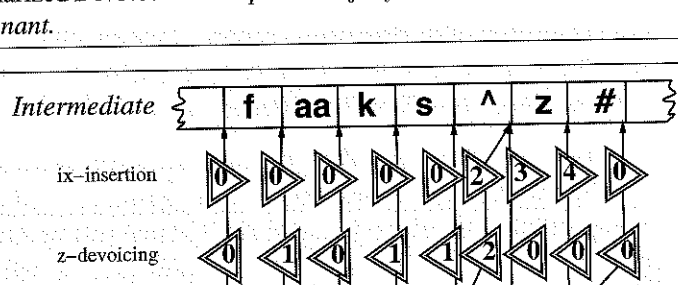


Figure 4.14 The transducer for the i-insertion rule 4.4 and the z-devoicing rule 4.5 run in parallel.

4.4 Advanced Issues in Computational Phonology

Harmony

Rules like flapping, i-insertion, and z-devoicing are relatively simple as phonological rules go. In this section we turn to the use of the two-level or finite-state model of phonology to model more sophisticated phenomena; this section will be easier to follow if the reader has some knowledge of phonology. The Yawelmani dialect of Yokuts is a Native American language spoken in California with a complex phonological system. In particular, there are three phonological rules related to the realization of vowels that had to be ordered in traditional phonology and whose interaction thus demonstrates a complicated use of finite-state phonology. These rules were first drawn up in the

traditional Chomsky and Halle (1968) format by Kisseberth (1969) following the field work of Newman (1944).

First, Yokuts (like many other languages including for example Turkish and Hungarian) has a phonological phenomenon called **vowel harmony**. VOWEL HARMONY Vowel harmony is a process in which a vowel changes its form to look like a neighboring vowel. In Yokuts, a suffix vowel changes its form to agree in backness and roundness with the preceding stem vowel. That is, a front vowel like /i/ will appear as a backvowel [u] if the stem vowel is /u/ (examples are taken from Cole and Kisseberth (1995):[6]

Lexical		Surface	Gloss
dub+hin	→	dubhun	"tangles, non-future"
xil+hin	→	xilhin	"leads by the hand, non-future"
bok'+al	→	bok'ol	"might eat"
xat'+al	→	xat'al	"might find"

This Harmony rule has another constraint: it only applies if the suffix vowel and the stem vowel are of the same height. Thus /u/ and /i/ are both high, while /o/ and /a/ are both low.

The second relevant rule, Lowering, causes long high vowels to become low; thus /u:/ becomes [o:] in the first example below:

Lexical		Surface	Gloss
ʔu:t'+it	→	ʔo:t'ut	"steal, passive aorist"
mi:k'+it	→	me:k'+it	"swallow, passive aorist"

The third rule, Shortening, shortens long vowels if they occur in closed syllables:

Lexical		Surface
s:ap+hin	→	saphin
sudu:k+hin	→	sudokhun

The Yokuts rules must be ordered, just as the i-insertion and z-devoicing rules had to be ordered. Harmony must be ordered before Lowering because the /u:/ in the lexical form /ʔu:t'+it/ causes the /i/ to become [u] before it lowers in the surface form [ʔo:t'ut]. Lowering must be ordered before Shortening because the /u:/ in /sudu:k+hin/ lowers to [o]; if it was ordered after shortening it would appear on the surface as [u].

Goldsmith (1993) and Lakoff (1993) independently observed that the Yokuts data could be modeled by something like a transducer; Karttunen

[6] For purposes of simplifying the explanation, this account ignores some parts of the system such as vowel underspecification (Archangeli, 1984).

(1998) extended the argument, showing that the Goldsmith and Lakoff constraints could be represented either as a cascade of three rules in series, or in the two-level formalism as three rules in parallel; Figure 4.15 shows the two architectures. Just as in the two-level examples presented earlier, the rules work by referring sometimes to the lexical context, sometimes to the surface context; writing the rules is left as Exercise 4.10 for the reader.

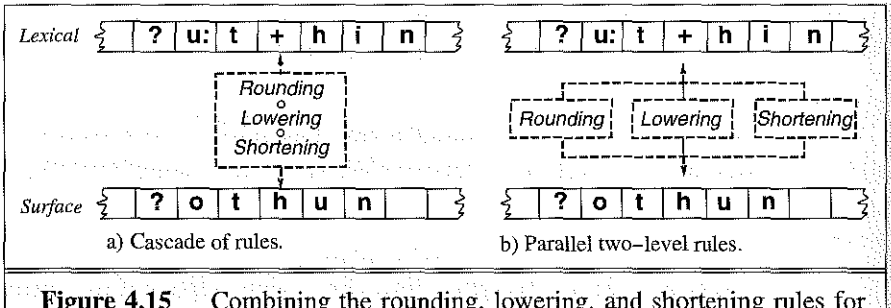


Figure 4.15 Combining the rounding, lowering, and shortening rules for Yawelmani Yokuts.

Templatic Morphology

Finite-state models of phonology/morphology have also been proposed for the templatic (non-concatenative) morphology (discussed on page 60) common in Semitic languages like Arabic, Hebrew, and Syriac. McCarthy (1981) proposed that this kind of morphology could be modeled by using different levels of representation that Goldsmith (1976) had called **tiers**. Kay (1987) proposed a computational model of these tiers via a special transducer which reads four tapes instead of two, as in Figure 4.16.

TIERS

The tricky part here is designing a machine which aligns the various strings on the tapes in the correct way; Kay proposed that the binyan tape could act as a sort of guide for alignment. Kay's intuition has led to a number of more fully worked out finite-state models of Semitic morphology such as Beesley's (1996) model for Arabic and Kiraz's (1997) model for Syriac.

The more recent work of Kornai (1991) and Bird and Ellison (1994) showed how one-tape automata (i.e. finite-state automata rather than four-tape or even two-tape transducers) could be used to model templatic morphology and other kinds of phenomena that are handleed with the tier-based **autosegmental** representations of Goldsmith (1976).

AUTOSEGMENTAL

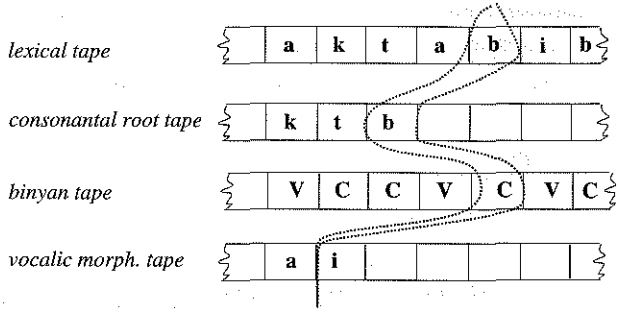


Figure 4.16 A finite-state model of templatic ("non-concatenative") morphology. Modified from Kay (1987) and Sproat (1993).

Optimality Theory

In a traditional phonological derivation, we are given an underlying lexical form and a surface form. The phonological system then consists of one component: a sequence of rules which map the underlying form to the surface form. **Optimality Theory** (**OT**) (Prince and Smolensky, 1993) offers an alternative way of viewing phonological derivation, based on two functions (GEN and EVAL) and a set of ranked violable constraints (CON). Given an underlying form, the GEN function produces all imaginable surface forms, even those which couldn't possibly be a legal surface form for the input. The EVAL function then applies each constraint in CON to these surface forms in order of constraint rank. The surface form which best meets the constraints is chosen.

OPTIMALITY THEORY

OT

A constraint in OT represents a wellformedness constraint on the surface form, such as a phonotactic constraint on what segments can follow each other, or a constraint on what syllable structures are allowed. A constraint can also check how **faithful** the surface form is to the underlying form.

FAITHFUL

Let's turn to our favorite complicated language, Yawelmani, for an example.[7] In addition to the interesting vowel harmony phenomena discussed above, Yawelmani has a phonotactic constraints that rules out sequences of consonants. In particular three consonants in a row (CCC) are not allowed to occur in a surface word. Sometimes, however, a word contains two consecutive morphemes such that the first one ends in two consonants and the second one starts with one consonant (or vice versa). What does the lan-

[7] The following explication of OT via the Yawelmani example draws heavily from Archangeli (1997) and a lecture by Jennifer Cole at the 1999 LSA Linguistic Institute.

guage do to solve this problem? It turns out that Yawelmani either deletes one of the consonants or inserts a vowel in between.

For example, if a stem ends in a C, and its suffix starts with CC, the first C of the suffix is deleted ("+" here means a morpheme boundary):

C-deletion $C \rightarrow \varepsilon / C + \_\_ C$ (4.8)

Here is an example where the CCVC "passive consequent adjunctive" morpheme hne:l (actually the underlying form is /hnil/) drops the initial C if the previous morpheme ends in two consonants (and an example where it doesn't, for comparison):

underlying morphemes	gloss
diyel-ne:l-aw	"guard - passive consequent adjunctive - locative"
cawa-hne:l-aw	"shout - passive consequent adjunctive - locative"

If a stem ends in CC and the suffix starts with C, the language instead inserts a vowel to break up the first two consonants:

V-insertion $\varepsilon \rightarrow V / C \_\_ C + C$ (4.9)

Here are some examples in which an i is inserted into the roots ʔilk- "sing" and the roots logw- "pulverize" only when they are followed by a C-initial suffix like -hin, "past", not a V-initial suffix like -en, "future":

surface form	gloss
ʔilik-hin	"sang"
ʔilken	"will sing"
logiwhin	"pulverized"
logwen	"will pulverize"

Kisseberth (1970) suggested that it was not a coincidence that Yawelmani had these particular two rules (and for that matter other related deletion rules that we haven't presented). He noticed that these rules were functionally related; in particular, they all are ways of avoiding three consonants in a row. Another way of stating this generalization is to talk about syllable structure. Yawelmani syllables are only allowed to be of the form CVC or CV (where C means a consonant and V means a vowel). We say that languages like Yawelmani don't allow **complex onsets** or **complex codas**. From the point of view of syllabification, then, these insertions and deletions all happen so as to allow Yawelmani words to be properly syllabified. Since CVCC syllables aren't allowed on the surface, CVCC roots must be **resyllabified** when they appear on the surface. For example, here are the syllabifications

COMPLEX ONSET
COMPLEX CODA
RESYLLABIFIED

of the Yawelmani words we have discussed and some others; note, for example, that the surface syllabification of the CVCC syllables moves the final consonant to the beginning of the next syllable:

underlying morphemes	surface syllabification	gloss
ʔilk-en	ʔil.ken	"will sing"
logw-en	log.wen	"will pulverize"
logw-hin	lo.giw.hin	"will pulverize"
xat-en	xa.ten	"will eat"
diyel-hnil-aw	di.yel.ne:.law	"ask - pass. cons. adjunct. - locative"

Here's where Optimality Theory comes in. The basic idea in Optimality Theory is that the language has various constraints on things like syllable structure. These constraints generally apply to the surface form. One such constraint, *COMPLEX, says "No complex onsets or codas". Another class of constraints requires the surface form to be identical to (faithful to) the underlying form. Thus FAITHV says "Don't delete or insert vowels" and FAITHC says "Don't delete or insert consonants". Given an underlying form, the GEN function produces all possible surface forms (i.e., every possible insertion and deletion of segments with every possible syllabification) and they are ranked by the EVAL function using these constraints. Figure 4.17 shows the architecture.

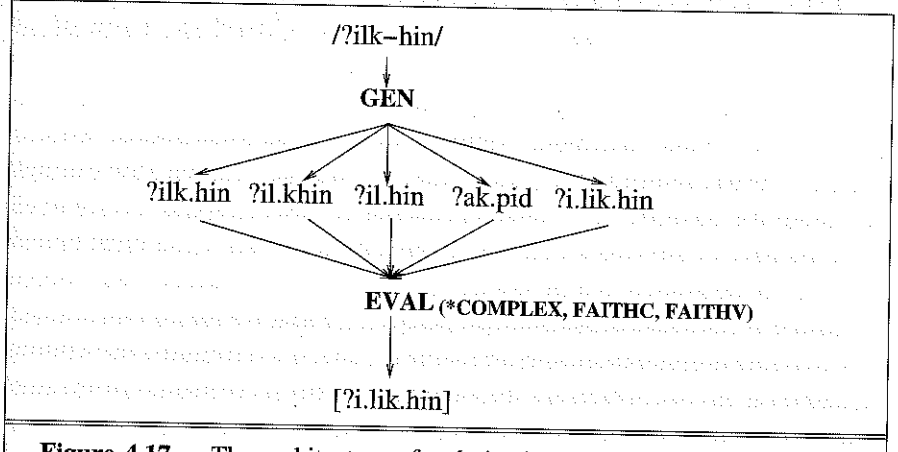


Figure 4.17 The architecture of a derivation in Optimality Theory (after Archangeli (1997)).

The EVAL function works by applying each constraint in ranked order; the optimal candidate is one which either violates no constraints, or violates

less of them than all the other candidates. This evaluation is usually shown on a **tableau** (plural **tableaux**). The top left-hand cell shows the input, the constraints are listed in order of rank across the top row, and the possible outputs along the left-most column. Although there are an infinite number of candidates, it is traditional to show only the ones which are 'close'; in the tableau below we have shown the output ?ak.pid just to make it clear that even very different surface forms are to be included. If a form violates a constraint, the relevant cell contains *; a !* indicates the fatal violation which causes a candidate to be eliminated. Cells for constraints which are irrelevant (since a higher-level constraint is already violated) are shaded.

TABLEAU

/?ilk-hin/	*COMPLEX	FAITHC	FAITHV
?ilk.hin	*!		
?il.khin	*!		
?il.hin		*!	
☞ ?i.lik.hin			*
?ak.pid		*!	

One appeal of Optimality Theoretic derivations is that the constraints are presumed to be cross-linguistic generalizations. That is all languages are presumed to have some version of faithfulness, some preference for simple syllables, and so on. Languages differ in how they rank the constraints; thus English, presumably, ranks FAITHC higher than *COMPLEX. (How do we know this?)

Can a derivation in Optimality Theory be implemented by finite-state transducers? Frank and Satta (1999), following the foundational work of Ellison (1994), showed that (1) if GEN is a regular relation (for example assuming the input doesn't contain context-free trees of some sort), and (2) if the number of allowed violations of any constraint has some finite bound, then an OT derivation can be computed by finite-state means. This second constraint is relevant because of a property of OT that we haven't mentioned: if two candidates violate exactly the same number of constraints, the winning candidate is the one which has the smallest number of violations of the relevant constraint.

One way to implement OT as a finite-state system was worked out by Karttunen (1998), following the above-mentioned work and that of Hammond (1997). In Karttunen's model, GEN is implemented as a finite-state transducer which is given an underlying form and produces a set of candidate forms. For example for the syllabification example above, GEN would

generate all strings that are variants of the input with consonant deletions or vowel insertions, and their syllabifications.

Each constraint is implemented as a filter transducer that lets pass only strings which meet the constraint. For legal strings, the transducer thus acts as the identity mapping. For example, *COMPLEX would be implemented via a transducer that mapped any input string to itself, unless the input string had two consonants in the onset or coda, in which case it would be mapped to null.

The constraints can then be placed in a cascade, in which higher-ranked constraints are simply run first, as suggested in Figure 4.18.

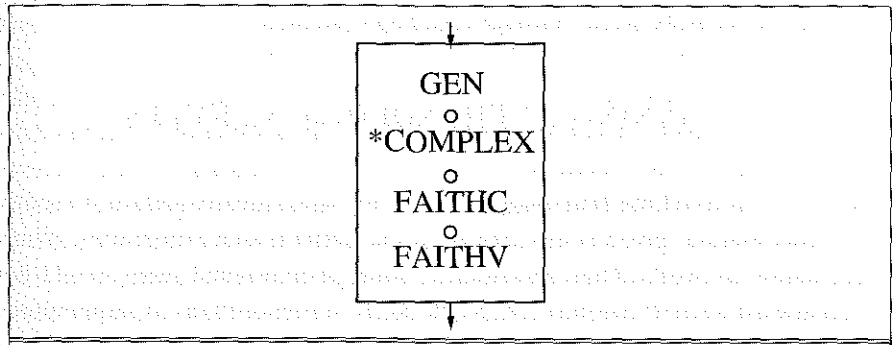


Figure 4.18 Version #1 ("merciless cascade") of Karttunen's finite-state cascade implementation of OT.

There is one crucial flaw with the cascade model in Figure 4.18. Recall that the constraints-transducers filter out any candidate which violates a constraint. But in many derivations, include the proper derivation of ?i.lik.hin, even the optimal form still violates a constraint. The cascade in Figure 4.17 would incorrectly filter it out, leaving no surface form at all! Frank and Satta (1999) and Hammond (1997) both point out that it is essential to only enforce a constraint if it does not reduce the candidate set to zero. Karttunen (1998) formalizes this intuition with the **lenient composition** operator. Lenient composition is a combination of regular composition and an operation called **priority union**. The basic idea is that if any candidates meet the constraint these candidates will be passed through the filter as usual. If no output meets the constraint, lenient composition retains *all* of the candidates. Figure 4.19 shows the general idea; the interested reader should see Karttunen (1998) for the details. Also see Tesar (1995, 1996), Fosler (1996), and Eisner (1997) for discussions of other computational issues in OT.

LENIENT COMPOSITION

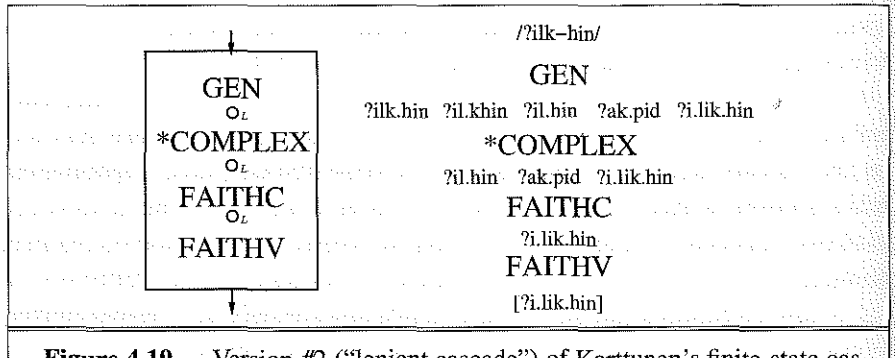


Figure 4.19 Version #2 ("lenient cascade") of Karttunen's finite-state cascade implementation of OT, showing a visualization of the candidate populations that would be passed through each FST constraint.

4.5 Machine Learning of Phonological Rules

MACHINE LEARNING

The task of a **machine learning** system is to automatically induce a model for some domain, given some data from the domain and, sometimes, other information as well. Thus a system to learn phonological rules would be given at least a set of (surface forms of) words to induce from. A **supervised** algorithm is one which is given the correct answers for some of this data, using these answers to induce a model which can generalize to new data it hasn't seen before. An **unsupervised** algorithm does this purely from the data. While unsupervised algorithms don't get to see the correct labels for the classifications, they can be given hints about the nature of the rules or models they should be forming. For example, the knowledge that the models will be in the form of automata is itself a kind of hint. Such hints are called a **learning bias**.

SUPERVISED

UNSUPERVISED

LEARNING BIAS

This section gives a very brief overview of some models of unsupervised machine learning of phonological rules; more details about machine learning algorithms will be presented throughout the book.

Ellison (1992) showed that concepts like the consonant and vowel distinction, the syllable structure of a language, and harmony relationships could be learned by a system based on choosing the model from the set of potential models which is the simplest. Simplicity can be measured by choosing the model with the minimum coding length, or the highest probability (we will define these terms in detail in Chapter 6). Daelemans et al. (1994) used the Instance-Based Generalization algorithm (Aha et al., 1991) to learn stress rule for Dutch; the algorithm is a supervised one which is

given a number of words together with their stress patterns, and which induces generalizations about the mapping from the sequences of light and heavy syllable type in the word (light syllables have no coda consonant; heavy syllables have one) to the stress pattern. Tesar and Smolensky (1993) show that a system which is given Optimality Theory constraints but not their ranking can learn the ranking from data via a simple greedy algorithm.

Johnson (1984) gives one of the first computational algorithms for phonological rule induction. His algorithm works for rules of the form

(4.10) $a \rightarrow b/C$

where C is the feature matrix of the segments around a. Johnson's algorithm sets up a system of constraint equations which C must satisfy, by considering both the positive contexts, i.e., all the contexts C_i in which a b occurs on the surface, as well as all the negative contexts C_j in which an a occurs on the surface. Touretzky et al. (1990) extended Johnson's insight by using the *version spaces* algorithm of Mitchell (1981) to induce phonological rules in their *Many Maps* architecture, which is similar to two-level phonology. Like Johnson's, their system looks at the underlying and surface realizations of single segments. For each segment, the system uses the version space algorithm to search for the proper statement of the context. The model also has a separate algorithm which handles harmonic effects by looking for multiple segmental changes in the same word, and is more general than Johnson's in dealing with epenthesis and deletion rules.

The algorithm of Gildea and Jurafsky (1996) was designed to induce transducers representing two-level rules of the type we have discussed earlier. Like the algorithm of Touretzky et al. (1990), Gildea and Jurafsky's algorithm was given sets of pairings of underlying and surface forms. The algorithm was based on the OSTIA (Oncina et al., 1993) algorithm, which is a general learning algorithm for a subtype of finite-state transducers called **subsequential transducers**. By itself, the OSTIA algorithm was too general to learn phonological transducers, even given a large corpus of underlying-form/surface-form pairs. Gildea and Jurafsky then augmented the domain-independent OSTIA system with three kinds of learning biases which are specific to natural language phonology; the main two are **Faithfulness** (underlying segments tend to be realized similarly on the surface), and **Community** (similar segments behave similarly). The resulting system was able to learn transducers for flapping in American English, or German consonant devoicing.

Finally, many learning algorithms for phonology are probabilistic. For

example Riley (1991) and Withgott and Chen (1993) proposed a decision-tree approach to segmental mapping. A decision tree is induced for each segment, classifying possible realizations of the segment in terms of contextual factors such as stress and the surrounding segments. Decision trees and probabilistic algorithms in general will be defined in Chapters 5 and 6.

4.6 Mapping Text to Phones for TTS

Dearest creature in Creation
Studying English pronunciation
I will teach you in my verse
Sounds like corpse, corps, horse and worse.
It will keep you, Susy, busy,
Make your head with heat grow dizzy
...
River, rival; tomb, bomb, comb;
Doll and roll, and some and home.
Stranger does not rime with anger
Neither does devour with clangour.
...

G.N. Trenite (1870-1946) *The Chaos*, reprinted in Witten (1982).

Now that we have learned the basic inventory of phones in English and seen how to model phonological rules, we are ready to study the problem of mapping from an orthographic or text word to its pronunciation.

Pronunciation Dictionaries

An important component of this mapping is a **pronunciation dictionary**. These dictionaries are actually used in both ASR and TTS systems, although because of the different needs of these two areas the contents of the dictionaries are somewhat different.

The simplest pronunciation dictionaries just have a list of words and their pronunciations:

Word	Pronunciation	Word	Pronunciation
cat	[kæt]	goose	[gus]
cats	[kæts]	geese	[gis]
pig	[pɪg]	hedgehog	['hɛdʒ.hɔg]
pigs	[pɪgz]	hedgehogs	['hɛdʒ.hɔgz]
fox	[fɑx]		
foxes	['fɑk.sɪz]		

Three large, commonly-used, on-line pronunciation dictionaries in this format are PRONLEX, CMUdict, and CELEX. These are used for speech recognition and can also be adapted for use in speech synthesis. The PRONLEX dictionary (LDC, 1995) was designed for speech recognition applications and contains pronunciations for 90,694 wordforms. It covers all the words used in many years of the Wall Street Journal, as well as the Switchboard Corpus. The CMU Pronouncing Dictionary was also developed for ASR purposes and has pronunciations for about 100,000 wordforms. The CELEX dictionary (Celex, 1993) includes all the words in the Oxford Advanced Learner's Dictionary (1974) (41,000 lemmata) and the Longman Dictionary of Contemporary English (1978) (53,000 lemmata), in total it has pronunciations for 160,595 wordforms. Its pronunciations are British while the other two are American. Each dictionary uses a different phone set; the CMU and PRONLEX phonesets are derived from the ARPAbet, while the CELEX dictionary is derived from the IPA. All three represent three levels of stress: primary stress, secondary stress, and no stress. Figure 4.20 shows the pronunciation of the word *armadillo* in all three dictionaries.

Dictionary	Pronunciation	IPA Version
Pronlex	+arm.xd'Il.o	[ˌɑrmə'dɪloʊ]
CMU	AA2 R M AH0 D IH1 L OW0	[ˌɑrm̲ʌ'dɪlo̲ʊ̲]
CELEX	"#-m@-'dI-l5	[ɑː.mə.'dɪ.ɫəʊ]

Figure 4.20 The pronunciation of the word *armadillo* in three dictionaries. Rather than explain special symbols, we have given an IPA equivalent for each pronunciation. The CMU dictionary represents unstressed vowels ([ə], [ɨ], etc.) by giving a 0 stress level to the vowel. We represented this by underlining in the IPA form. Note the r-dropping and use of the [əu] rather than [ou] vowel in the British CELEX pronunciation.

Often two distinct words are spelled the same (they are **homographs**) but pronounced differently. For example the verb *wind* ("You need to wind this up more neatly") is pronounced [waɪnd] while the noun *wind* ("blow,

blow, thou winter wind") is pronounced [wɪnd]. This is essential for TTS applications (since in a given context the system needs to say one or the other) but for some reason is usually ignored in current speech recognition systems. Printed pronunciation dictionaries give distinct pronunciations for each part-of-speech; CELEX does as well. Since they were designed for ASR, Pronlex and CMU, although they give two pronunciations for the form *wind*, don't specify which one is used for which part-of-speech.

Dictionaries often don't include many proper names. This is a serious problem for many applications; Liberman and Church (1992) report that 21% of the word tokens in their 33-million-word 1988 AP newswire corpus were names. Furthermore, they report that a list obtained in 1987 from the Donnelly marketing organization contains 1.5 million names (covering 72 million households in the United States). But only about 1000 of the 52477 lemmas in CELEX (which is based on traditional dictionaries) are proper names. By contrast Pronlex includes 20,000 names; this is still only a small fraction of the 1.5 million. Very few dictionaries give pronunciations for entries like *Dr.*, which as Liberman and Church (1992) point out can be "doctor" or "drive", or *2/3*, which can be "two thirds" or "February third" or "two slash three".

No dictionaries currently have good models for the pronunciation of function words (*and*, *I*, *a*, *the*, *of*, etc.). This is because the variation in these words due to phonetic context is so great. Usually the dictionaries include some simple baseform (such as [ðɪ] for *the*) and use other algorithms to derive the variation due to context; Chapter 5 will treat the issue of modeling contextual pronunciation variation for words of this sort.

One significant difference between TTS and ASR dictionaries is that TTS dictionaries do not have to represent dialectal variation; thus where a very accurate ASR dictionary needs to represent both pronunciations of *either* and *tomato*, a TTS dictionary can choose one.

Beyond Dictionary Lookup: Text Analysis

Mapping from text to phones relies on the kind of pronunciation dictionaries we talked about in the last section. As we suggested before, one way to map text-to-phones would be to look up each word in a pronunciation dictionary and read the string of phones out of the dictionary. This method would work fine for any word that we can put in the dictionary in advance. But as we saw in Chapter 3, it's not possible to represent every word in English (or any other language) in advance. Both speech synthesis and speech recognition

systems need to be able to guess at the pronunciation of words that are not in their dictionary. This section will first examine the kinds of words that are likely to be missing in a pronunciation dictionary, and then show how the finite-state transducers of Chapter 3 can be used to model the basic task of text-to-phones. Chapter 5 will introduce variation in pronunciation and introduce probabilistic techniques for modeling it.

Three of the most important cases where we cannot rely on a word dictionary involve **names**, **morphological productivity**, and **numbers**. As a brief example, we arbitrarily selected a brief (561 word) movie review that appeared in the July 17, 1998 issue of the New York Times. The review, of Vincent Gallo's "Buffalo '66", was written by Janet Maslin. Here's the beginning of the article:

> In Vincent Gallo's "Buffalo '66," Billy Brown (Gallo) steals a blond kewpie doll named Layla (Christina Ricci) out of her tap dancing class and browbeats her into masquerading as his wife at a dinner with his parents. Billy hectors, cajoles and tries to bribe Layla. ("You can eat all the food you want. Just make me look good.") He threatens both that he will kill her and that he won't be her best friend. He bullies her outrageously but with such crazy brio and jittery persistence that Layla falls for him. Gallo's film, a deadpan original mixing pathos with bravado, works on its audience in much the same way.

We then took two large commonly-used on-line pronunciation dictionaries; the PRONLEX dictionary, that contains pronunciations for 90,694 wordforms and includes coverage of many years of the Wall Street Journal, as well as the Switchboard Corpus, and the larger CELEX dictionary, which has pronunciations for 160,595 wordforms. The combined dictionaries have approximately 194,000 pronunciations. Of the 561 words in the movie review, 16 (3%) did not have pronunciations in these two dictionaries (not counting two hyphenated words, *baby-blue* and *hollow-eyed*). Here they are:

Names		**Inflected Names**	**Numbers**	**Other**
Aki	Gazzara	Gallo's	'66	c'mere
Anjelica	Kaurismaki			indie
Arquette	Kusturica			kewpie
Buscemi	Layla			sexpot
Gallo	Rosanna			

Some of these missing words can be found by increasing the dictionary size (for example Wells's (1990) definitive (but not on-line) pronunciation

dictionary of English does have *sexpot* and *kewpie*). But the rest need to generated on-line.

Names are a large problem for pronunciation dictionaries. It is difficult or impossible to list in advance all proper names in English; furthermore they may come from any language, and may have variable spellings. Most potential applications for TTS or ASR involve names; for example names are essentially in telephony applications (directory assistance, call routing). Corporate names are important in many applications and are created constantly (*CoComp, Intel, Cisco*). Medical speech applications (such as transcriptions of doctor-patient interviews) require pronunciations of names of pharmaceuticals; there are some off-line medical pronunciation dictionaries but they are known to be extremely inaccurate (Markey and Ward, 1997). Recall the figure of 1.5 million names mentioned above, and Liberman and Church's (1992) finding that 21% of the word tokens in their 33 million word 1988 AP newswire corpus were names.

Morphology is a particular problem for many languages other than English. For languages with very productive morphology it is computationally infeasible to represent every possible word; recall this Turkish example:

(4.11) uygarlaştıramadıklarımızdanmışsınızcasına

uygar	*+laş*	*+tır*	*+ama*	*+dık*	*+lar*	*+ımız*
civilized	+BEC	+CAUS	+NEGABLE	+PPART	+PL	+P1PL

+dan	*+mış*	*+sınız*	*+casına*
+ABL	+PAST	+2PL	+AsIf

"(behaving) as if you are among those whom we could not civilize/cause to become civilized"

Even a language as similar to English as German has greater ability to create words; Sproat et al. (1998) note the spontaneously created German example *Unerfindlichkeitsunterstellung* ("allegation of incomprehensibility").

But even in English, morphologically simple though it is, morphological knowledge is necessary for pronunciation modeling. For example names and acronyms are often inflected (*Gallo's, IBM's, DATs, Syntex's*) as are new words (*faxes, indies*). Furthermore, we can't just add *s* to the pronunciation of the uninflected forms, because as the last section showed, the possessive *-'s* and plural *-s* suffix in English are pronounced differently in different contexts; *Syntex's* is pronounced [sɪntɛksɨz], *faxes* is pronounced [fæksɨz], *IBM's* is pronounced [aɪbijɛmz], and *DATs* is pronounced [dæts].

Finally, pronouncing numbers is a particularly difficult problem. The *'66* in *Buffalo '66* is pronounced [sɪkstisɪks] not [sɪkssɪks]. The most natural

way to pronounce the phone number "947-2020" is probably "nine"-"four"-"seven"-"twenty"-"twenty" rather than "nine"-"four"-"seven"-"two"-"zero"-"two"-"zero". Liberman and Church (1992) note that there are five main ways to pronounce a string of digits (although others are possible):

- **Serial:** Each digit is pronounced separately—*8765* is "eight seven six five".
- **Combined:** The digit string is pronounced as a single integer, with all position labels read out—"eight thousand seven hundred sixty five".
- **Paired:** Each pair of digits is pronounced as an integer; if there is an odd number of digits the first one is pronounced by itself—"eighty-seven sixty-five".
- **Hundreds:** Strings of four digits can be pronounced as counts of hundreds—"eighty-seven hundred (and) sixty-five".
- **Trailing Unit:** Strings that end in zeros are pronounced serially until the last nonzero digit, which is pronounced followed by the appropriate unit—*8765000* is "eight seven six five thousand".

Pronunciation of numbers and these five methods are discussed further in Exercises 4.5 and 4.6.

An FST-based Pronunciation Lexicon

Early work in pronunciation modeling for text-to-speech systems (such as the seminal MITalk system Allen et al. (1987)) relied heavily on **letter-to-sound** rules. Each rule specified how a letter or combination of letters was mapped to phones; here is a fragment of such a rule-base from Witten (1982):

LETTER-TO-SOUND

Fragment	Pronunciation
-p-	[p]
-ph-	[f]
-phe-	[fi]
-phes-	[fiz]
-place-	[pleɪs]
-placi-	[pleɪsi]
-plement-	[plɪmɛnt]

Such systems consisted of a long list of such rules and a very small dictionary of exceptions (often function words such as *a, are, as, both, do, does*, etc.). More recent systems have completely inverted the algorithm, relying on very large dictionaries, with letter-to-sound rules only used for the small

number of words that are neither in the dictionary nor are morphological variants of words in the dictionary. How can these large dictionaries be represented in a way that allows for morphological productivity? Luckily, these morphological issues in pronunciation (adding inflectional suffixes, slight pronunciation changes at the juncture of two morphemes, etc.) are identical to the morphological issues in spelling that we saw in Chapter 3. Indeed, (Sproat, 1998b) and colleagues have worked out the use of transducers for text-to-speech. We might break down their transducer approach into five components:

1. an FST to represent the pronunciation of individual words and morphemes in the lexicon
2. FSAs to represent the possible sequencing of morphemes
3. individual FSTs for each pronunciation rule (for example expressing the pronunciation of *-s* in different contexts)
4. heuristics and letter-to-sound (LTS) rules/transducers used to model the pronunciations of names and acronyms
5. default letter-to-sound rules/transducers for any other unknown words

We will limit our discussion here to the first four components; those interested in letter-to-sound rules should see (Allen et al., 1987). These first components will turn out to be simple extensions of the FST components we saw in Chapter 3 and on page 110. The first is the representation of the lexical base form of each word; recall that base form means the uninflected form of the word. The previous base forms were stored in orthographic representation; we will need to augment each of them with the correct lexical phonological representation. Figure 4.21 shows the original and the updated lexical entries:

The second part of our FST system is the finite-state machinery to model morphology. We will give only one example: the nominal plural suffix *-s*. Figure 4.22 in Chapter 3 shows the automaton for English plurals, updated to handle pronunciation as well. The only change was the addition of the [s] pronunciation for the suffix, and ε pronunciations for all the morphological features.

We can compose the inflection FSA in Figure 4.22 with a transducer implementing the baseform lexicon in Figure 4.21 to produce an inflectionally-enriched lexicon that has singular and plural nouns. The resulting mini-lexicon is shown in Figure 4.23.

The lexicon shown in Figure 4.23 has two levels, an underlying or "lexical" level and an intermediate level. The only thing that remains is to add

Orthographic Lexicon	Lexicon
Regular Nouns	
cat	c|k a|æ t|t
fox	f|f o|ɑ x|ks
dog	d|d o|ɑ g|g
Irregular Singular Nouns	
goose	g|g oo|u s|s e|ɛ
Irregular Plural Nouns	
g o:e o:e s e	g|g oo|u:ee|i s|s e|ɛ

Figure 4.21 FST-based lexicon, extending the lexicon in the table on page 74 in Chapter 3. Each symbol in the lexicon is now a pair of symbols separated by "|", one representing the "orthographic" lexical entry and one the "phonological" lexical entry. The irregular plural *geese* also pre-specifies the contents of the intermediate tape ":ee|i".

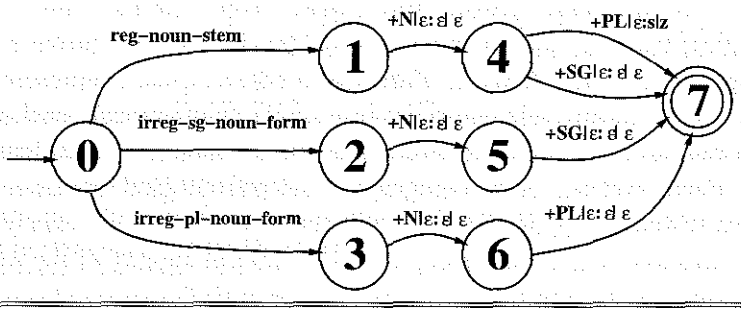


Figure 4.22 FST for the nominal singular and plural inflection. The automaton adds the morphological features [+N], [+PL], and [+SG] at the lexical level where relevant and also adds the plural suffix *s*|z (at the intermediate level). We will discuss below why we represent the pronunciation of *-s* as z rather than s.

transducers which apply spelling rules and pronunciation rules to map the intermediate level into the surface level. These include the various spelling rules discussed on page 77 and the pronunciation rules starting on page 105.

The lexicon and these phonological rules and the orthographic rules from Chapter 3 can now be used to map between a lexical representation (containing both orthographic and phonological strings) and a surface representation (containing both orthographic and phonological strings). As we saw in Chapter 3, this mapping can be run from surface to lexical form, or from lexical to surface form; Figure 4.24 shows the architecture. Recall that

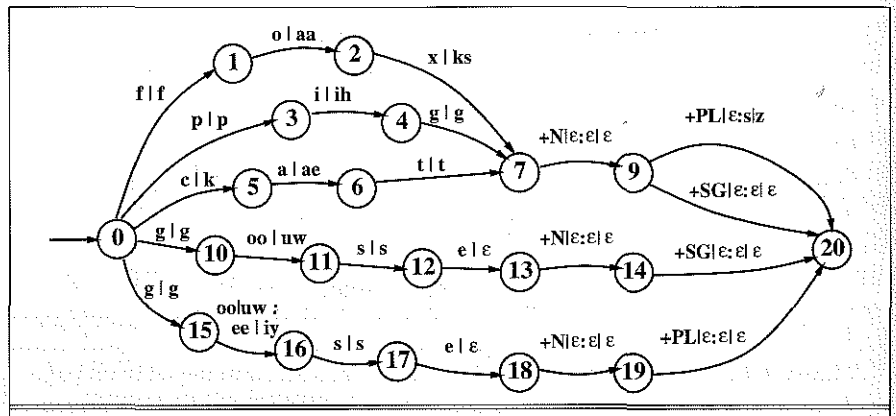


Figure 4.23 Mini-lexicon composing a transducer from the baseform lexicon of Figure 4.21 with the inflectional transducer of Figure 4.22.

the lexicon FST maps between the "lexical" level, with its stems and morphological features, and an "intermediate" level which represents a simple concatenation of morphemes. Then a host of FSTs, each representing either a single spelling rule constraint or a single phonological constraint, all run in parallel so as to map between this intermediate level and the surface level. Each level has both orthographic and phonological representations. For text-to-speech applications in which the input is a lexical form (e.g., for text generation, where the system knows the lexical identity of the word, its part-of-speech, its inflection, etc.), the cascade of FSTs can map from lexical form to surface pronunciation. For text-to-speech applications in which the input is a surface spelling (e.g., for "reading text out loud" applications), the cascade of FSTs can map from surface orthographic form to surface pronunciation via the underlying lexical form.

Finally let us say a few words about names and acronyms. Acronyms can be spelled with or without periods (*I.R.S.* or *IRS*). Acronyms with periods are usually pronounced by spelling them out ([aɪɑrɛs]). Acronyms that usually appear without periods (AIDS, ANSI, ASCAP) may either be spelled out or pronounced as a word; so AIDS is usually pronounced the same as the third-person form of the verb *aid*. Liberman and Church (1992) suggest keeping a small dictionary of the acronyms that are pronounced as words, and spelling out the rest. Their method for dealing with names begins with a dictionary of the pronunciations of 50,000 names, and then applies a small number of affix-stripping rules (akin to the Porter Stemmer of Chapter 3), rhyming heuristics, and letter-to-sound rules to increase the coverage.

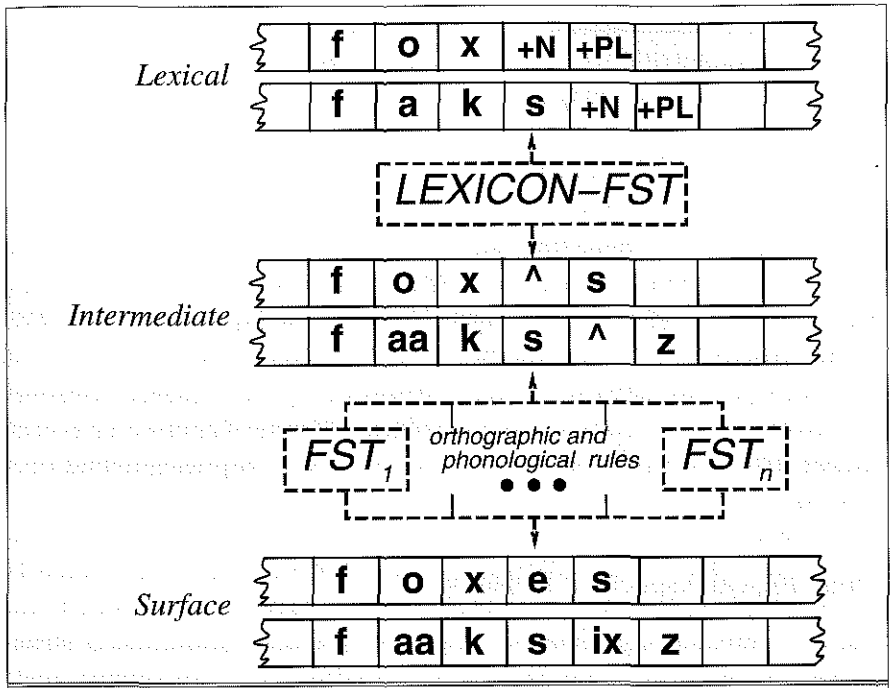


Figure 4.24 Mapping between the lexicon and surface form for orthography and phonology simultaneously. The system can be used to map from a lexical entry to its surface pronunciation or from surface orthography to surface pronunciation via the lexical entry.

Liberman and Church (1992) took the most frequent quarter million words in the Donnelly list. They found that the 50,000 word dictionary covered 59% of these 250,000 name tokens. Adding stress-neutral suffixes like *-s*, *-ville*, and *-son* (*Walters* = *Walter* + *s*, *Abelson* = *Abel* + *son*, *Lucasville* = *Lucas* + *ville*) increased the coverage to 84%. Adding name-name compounds (*Abdulhussein*, *Baumgaertner*) and rhyming heuristics increased the coverage to 89%. The rhyming heuristics used letter-to-sound rules for the beginning of the word and then found a rhyming word to help pronounce the end; so Plotsky was pronounced by using the LTS rule for *Pl-* and guessing *-otsky* from *Trotsky*. They then added a number of more complicated morphological rules (prefixes like *O'Brien*), stress-changing suffixes (*Adamovich*), suffix-exchanges (*Bierstadt* = *Bierbaum* - *baum* + *stadt*) and used a system of letter-to-sound rules for the remainder. This system was not implemented as an FST; Exercise 4.11 will address some of the issues in turning such a set of rules into an FST. Readers interested in further details about names,

acronyms and other unknown words should consult sources such as Liberman and Church (1992), Vitale (1991), and Allen et al. (1987).

4.7 PROSODY IN TTS

The orthography to phone transduction process just described produces the main component for the input to the part of a TTS system which actually generates the speech. Another important part of the input is a specification of the **prosody**. The term **prosody** is generally used to refer to aspects of a sentence's pronunciation which aren't described by the sequence of phones derived from the lexicon. Prosody operates on longer linguistic units than phones, and hence is sometimes called the study of **suprasegmental** phenomena.

PROSODY

SUPRASEGMENTAL

Phonological Aspects of Prosody

There are three main phonological aspects to prosody: **prominence**, **structure** and **tune**.

PROMINENCE

STRUCTURE

TUNE

As page 102 discussed, prominence is a broad term used to cover **stress** and **accent**. Prominence is a property of syllables, and is often described in a relative manner, by saying one syllable is more prominent than another. Pronunciation lexicons mark lexical stress; for example *table* has its stress on the first syllable, while *machine* has its stress on the second. Function words like *there*, *the* or *a* are usually unaccented altogether. When words are joined together, their accentual patterns combine and form a larger accent pattern for the whole utterance. There are some regularities in how accents combine. For example adjective-noun combinations like *new truck* are likely to have accent on the right word (*new *truck*), while noun-noun compounds like **tree surgeon* are likely to have accent on the left. In generally, however, there are many exceptions to these rules, and so accent prediction is quite complex. For example the noun-noun compound **apple cake* has the accent on the first word while the noun-noun compound *apple *pie* or *city *hall* both have the accent on the second word (Liberman and Sproat, 1992; Sproat, 1994, 1998a). Furthermore, rhythm plays a role in keeping the accented syllables spread apart a bit; thus *city *hall* and **parking lot* combine as **city hall *parking lot* (Liberman and Prince, 1977). Finally, the location of accent is very strongly affected by the discourse factors we will describe in Chapters 18 and 19; in particular new or focused words or phrases often receive accent.

STRESS

ACCENT

Sentences have prosodic structure in the sense that some words seem to group naturally together and some words seem to have a noticeable break or disjuncture between them. Often prosodic structure is described in terms of **prosodic phrasing**, meaning that an utterance has a prosodic phrase structure in a similar way to it having a syntactic phrase structure. For example, in the sentence *I wanted to go to London, but could only get tickets for France* there seems to be two main prosodic phrases, their boundary occurring at the comma. Commonly used terms for these larger prosodic units include **intonational phrase** or **IP** (Beckman and Pierrehumbert, 1986), **intonation unit** (Du Bois et al., 1983), and **tone unit** (Crystal, 1969). Furthermore, in the first phrase, there seems to be another set of lesser prosodic phrase boundaries (often called **intermediate phrases**) that split up the words as follows *I wanted | to go | to London.* The exact definitions of prosodic phrases and subphrases and their relation to syntactic phrases like clauses and noun phrases and semantic units have been and still are the topic of much debate (Chomsky and Halle, 1968; Langendoen, 1975; Streeter, 1978; Hirschberg and Pierrehumbert, 1986; Selkirk, 1986; Nespor and Vogel, 1986; Croft, 1995; Ladd, 1996; Ford and Thompson, 1996; Ford et al., 1996). Despite these complications, algorithms have been proposed which attempt to automatically break an input text sentence into intonational phrases. For example Wang and Hirschberg (1992), Ostendorf and Veilleux (1994), Taylor and Black (1998), and others have built statistical models (incorporating probabilistic predictors such as the CART-style decision trees to be defined in Chapter 5) for predicting intonational phrase boundaries based on such features as the parts of speech of the surrounding words, the length of the utterance in words and seconds, the distance of the potential boundary from the beginning or ending of the utterance, and whether the surrounding words are accented.

PROSODIC PHRASING

INTONATIONAL PHRASE

IP

INTERMEDIATE PHRASE

Two utterances with the same prominence and phrasing patterns can still differ prosodically by having different **tunes**. Tune refers to the intonational melody of an utterance. Consider the utterance *oh, really*. Without varying the phrasing or stress, it is still possible to have many variants of this by varying the intonational tune. For example, we might have an excited version *oh, really!* (in the context of a reply to a statement that you've just won the lottery); a sceptical version *oh, really?*—in the context of not being sure that the speaker is being honest; to an angry *oh, really!* indicating displeasure. Intonational tunes can be broken into component parts, the most important of which is the **pitch accent**. Pitch accents occur on stressed syllables and form a characteristic pattern in the F0 contour (as explained below).

PITCH ACCENT

Depending on the type of pattern, different effects (such as those just outlined above) can be produced. A popular model of pitch accent classification is the Pierrehumbert or ToBI model (Pierrehumbert, 1980; Silverman et al., 1992), which says there are five pitch accents in English, which are made from combining two simple tones (high **H**, and low **L**) in various ways. A **H+L** pattern forms a fall, while a **L+H** pattern forms a rise. An asterisk (*) is also used to indicate which tone falls on the stressed syllable. This gives an inventory of **H*, L*, L+H*, L*+H, H+L*** (a sixth pitch accent **H*+L** which was present in early versions of the model was later abandoned). Our three examples of *oh, really* might be marked with the accents **L+H*, L*+H and L*** respectively. In addition to pitch accents, this model also has two phrase accents **L-** and **H-** and two boundary tones **L%** and **H%**, which are used at the ends of phrases to control whether the intonational tune rises or falls.

Other intonational modals differ from ToBI by not using discrete phonemic classes for intonation accents. For example the Tilt (Taylor, 2000) and Fujisaki models (Fujisaki and Ohno, 1997) use continuous parameters rather than discrete categories to model pitch accents. These researchers argue that while the discrete models are often easier to visualize and work with, continuous models may be more robust and more accurate for computational purposes.

Phonetic or Acoustic Aspects of Prosody

The three phonological factors interact and are realized by a number of different phonetic or acoustic phenomena. Prominent syllables are generally louder and longer that non-prominent syllables. Prosodic phrase boundaries are often accompanied by pauses, by lengthening of the syllable just before the boundary, and sometimes lowering of pitch at the boundary. Intonational tune is manifested in the fundamental frequency (F0) contour.

Prosody in Speech Synthesis

A major task for a TTS system is to generate appropriate linguistic representations of prosody, and from them generate appropriate acoustic patterns which will be manifested in the output speech waveform. The output of a TTS system with such a prosodic component is a sequence of phones, each of which has a duration and an F0 (pitch) value. The duration of each phone is dependent on the phonetic context (see Chapter 7). The F0 value

is influenced by the factors discussed above, including the lexical stress, the accented or focused element in the sentence, and the intonational tune of the utterance (for example a final rise for questions). Figure 4.25 shows some sample TTS output from the FESTIVAL (Black et al., 1999) speech synthesis system for the sentence *Do you really want to see all of it?*. This output, together with the F0 values shown in Figure 4.26 would be the input to the **waveform synthesis** component described in Chapter 7. The durations here are computed by a CART-style decision tree (Riley, 1992).

				H*												L*		L- H%			
do		you		really				want				to		see		all		of		it	
d	uw	y	uw	r	ih	l	iy	w	aa	n	t	t	ax	s	iy	ao	l	ah	v	ih	t
110	110	50	50	75	64	57	82	57	50	72	41	43	47	54	130	76	90	44	62	46	220

Figure 4.25 Output of the FESTIVAL (Black et al., 1999) generator for the sentence *Do you really want to see all of it?* The exact intonation contour is shown in Figure 4.26. Thanks to Paul Taylor for this figure.

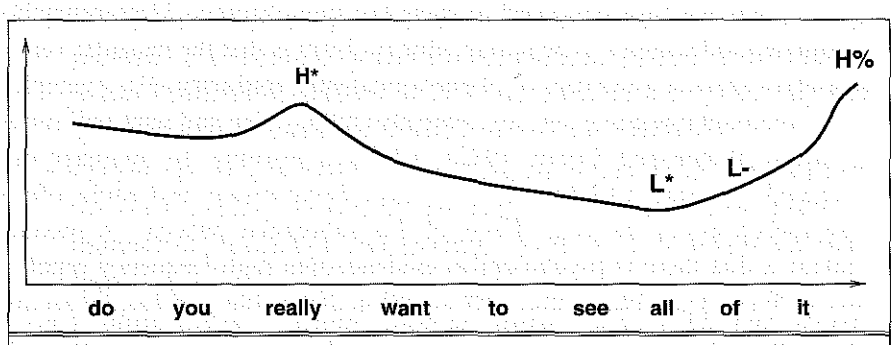


Figure 4.26 The F0 contour for the sample sentence generated by the FESTIVAL synthesis system in Figure 4.25, thanks to Paul Taylor.

As was suggested above, determining the proper prosodic pattern for a sentence is difficult, as real-world knowledge and semantic information is needed to know which syllables to accent, and which tune to apply. This sort of information is difficult to extract from the text and hence prosody modules often aim to produce a "neutral declarative" version of the input text, which assume the sentence should be spoken in a default way with no reference to discourse history or real-world events. This is one of the main reasons why intonation in TTS often sounds "wooden".

4.8 HUMAN PROCESSING OF PHONOLOGY AND MORPHOLOGY

Chapter 3 suggested that productive morphology plays a psychologically real role in the human lexicon. But we stopped short of a detailed model of how the morphology might be represented. Now that we have studied phonological structure and phonological learning, we return to the psychological question of the representation of morphological/phonological knowledge.

One view of human morphological or phonological processing might be that it distinguishes productive, regular morphology from irregular or exceptional morphology. Under this view, the regular past tense morpheme *-ed*, for example, could be mentally represented as a rule which would be applied to verbs like *walk* to produce *walked.* Irregular past tense verbs like *broke*, *sang*, and *brought*, on the other hand, would simply be stored as part of a lexical representation, and the rule wouldn't apply to these. Thus this proposal strongly distinguishes representation via *rules* from representation via lexical *listing*.

This proposal seems sensible, and is indeed identical to the transducer-based models we have presented in these last two chapters. Unfortunately, this simple model seems to be wrong, One problem is that the irregular verbs themselves show a good deal of phonological **subregularity**. For example, the ɪ/æ alternation relating *ring* and *rang* also relates *sing* and *sang* and *swim* and *swam* (Bybee and Slobin, 1982). Children learning the language often extend this pattern to incorrectly produce *bring-brang*, and adults often make speech errors showing effects of this subregular pattern. A second problem is that there is psychological evidence that high-frequency regular inflected forms (*needed, covered*) are stored in the lexicon just like the stems *cover* and *need* (Losiewicz, 1992). Finally, word and morpheme frequency in general seems to play an important role in human processing.

SUBREGULARITY

Arguments like these led to "data-driven" models of morphological learning and representation, which essentially store all the inflected forms they have seen. These models generalize to new forms by a kind of analogy; regular morphology is just like subregular morphology but acquires rule-like trappings simply because it occurs more often. Such models include the computational **connectionist** or **Parallel Distributed Processing** model of Rumelhart and McClelland (1986) and subsequent improvements (Plunkett and Marchman, 1991; MacWhinney and Leinbach, 1991) and the similar **network** model of Bybee (1985, 1995). In these models, the behavior of regular morphemes like *-ed* **emerges** from its frequent interaction with other

CONNECTIONIST PARALLEL DISTRIBUTED PROCESSING

forms. Proponents of the rule-based view of morphology such as Pinker and Prince (1988), Marcus et al. (1995), and others, have criticized the connectionist models and proposed a compromise **dual processing** model, in which regular forms like *-ed* are represent as symbolic rules, but subregular examples (*broke, brought*) are represented by connectionist-style pattern associators. This debate between the connectionist and dual processing models has deep implications for mental representation of all kinds of regular rule-based behavior and is one of the most interesting open questions in human language processing. Chapter 7 will briefly discuss connectionist models of human speech processing; readers who are further interested in connectionist models should consult the references above and textbooks like Anderson (1995).

4.9 SUMMARY

This chapter has introduced many of the important notions we need to understand spoken language processing. The main points are as follows:

- We can represent the pronunciation of words in terms of units called **phones**. The standard system for representing phones is the **International Phonetic Alphabet** or **IPA**. An alternative English-only transcription system that uses ASCII letters is the **ARPAbet**.
- Phones can be described by how they are produced **articulatorily** by the vocal organs; consonants are defined in terms of their **place** and **manner** of articulation and **voicing**, vowels by their **height** and **backness**.
- A **phoneme** is a generalization or abstraction over different phonetic realizations. **Allophonic rules** express how a phoneme is realized in a given context.
- **Transducers** can be used to model phonological rules just as they were used in Chapter 3 to model spelling rules. **Two-level morphology** is a theory of morphology/phonology which models phonological rules as finite-state **well-formedness constraints** on the mapping between lexical and surface form.
- **Pronunciation dictionaries** are used for both text-to-speech and automatic speech recognition. They give the pronunciation of words as strings of phones, sometimes including syllabification and stress. Most on-line pronunciation dictionaries have on the order of 100,000 words but still lack many names, acronyms, and inflected forms.

- The **text-analysis** component of a text-to-speech system maps from orthography to strings of phones. This is usually done with a large dictionary augmented with a system (such as a transducer) for handling productive morphology, pronunciation changes, names, numbers, and acronyms.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The major insights of articulatory phonetics date to the linguists of 800–150 B.C. India. They invented the concepts of place and manner of articulation, worked out the glottal mechanism of voicing, and understood the concept of assimilation. European science did not catch up with the Indian phoneticians until over 2000 years later, in the late 19th century. The Greeks did have some rudimentary phonetic knowledge; by the time of Plato's *Theaetetus* and *Cratylus*, for example, they distinguished vowels from consonants, and stop consonants from continuants. The Stoics developed the idea of the syllable and were aware of phonotactic constraints on possible words. An unknown Icelandic scholar of the twelfth century exploited the concept of the phoneme, proposed a phonemic writing system for Icelandic, including diacritics for length and nasality. But his text remained unpublished until 1818 and even then was largely unknown outside Scandinavia (Robins, 1967). The modern era of phonetics is usually said to have begun with Sweet, who proposed what is essentially the phoneme in his *Handbook of Phonetics* (1877). He also devised an alphabet for transcription and distinguished between *broad* and *narrow* transcription, proposing many ideas that were eventually incorporated into the IPA. Sweet was considered the best practicing phonetician of his time; he made the first scientific recordings of languages for phonetic purposes, and advanced the start of the art of articulatory description. He was also infamously difficult to get along with, a trait that is well captured in the stage character that George Bernard Shaw modeled after him: Henry Higgins. The phoneme was first named by the Polish scholar Baudouin de Courtenay, who published his theories in 1894.

The idea that phonological rules could be modeled as regular relations dates to Johnson (1972), who showed that any phonological system that didn't allow rules to apply to their own output (i.e., systems that did not have recursive rules) could be modeled with regular relations (or finite-state transducers). Virtually all phonological rules that had been formulated at

the time had this property (except some rules with integral-valued features, like early stress and tone rules). Johnson's insight unfortunately did not attract the attention of the community, and was independently discovered by Roland Kaplan and Martin Kay; see Chapter 3 for the rest of the history of two-level morphology. Karttunen (1993) gives a tutorial introduction to two-level morphology that includes more of the advanced details than we were able to present here.

Readers interested in phonology should consult (Goldsmith, 1995) as a reference on phonological theory in general and Archangeli and Langendoen (1997) on Optimality Theory.

Two classic text-to-speech synthesis systems are described in Allen et al. (1987) (the *MITalk* system) and Sproat (1998b) (the Bell Labs system). The pronunciation problem in text-to-speech synthesis is an ongoing research area; much of the current research focuses on prosody. Interested readers should consult the proceedings of the main speech engineering conferences: *ICSLP* (the International Conference on Spoken Language Processing), *IEEE ICASSP* (the International Conference on Acoustics, Speech, and Signal Processing), and *EUROSPEECH*.

Students with further interest in transcription and articulatory phonetics should consult an introductory phonetics textbook such as Ladefoged (1993). Pullum and Ladusaw (1996) is a comprehensive guide to each of the symbols and diacritics of the IPA. Many phonetics papers of computational interest are to be found in the *Journal of the Acoustical Society of America (JASA)*, *Computer Speech and Language*, and *Speech Communication*.

EXERCISES

4.1 Find the mistakes in the IPA transcriptions of the following words:

a. "three" [ðri]
b. "sing" [sɪng]
c. "eyes" [aɪs]
d. "study" [stʊdi]
e. "though" [θoʊ]

f. "planning" [plɑnɪŋ]
g. "slight" [slit]

4.2 Translate the pronunciations of the following color words from the IPA into the ARPAbet (and make a note if you think you pronounce them differently than this!):

a. [rɛd]
b. [blu]
c. [grin]
d. ['jɛlou]
e. [blæk]
f. [waɪt]
g. ['ɔrɪndʒ]
h. ['pɝpl]
i. [pjus]
j. [toʊp]

4.3 Ira Gershwin's lyric for *Let's Call the Whole Thing Off* talks about two pronunciations of the word "either" (in addition to the tomato and potato example given at the beginning of the chapter. Transcribe Ira Gershwin's two pronunciations of "either" in IPA and in the ARPAbet.

4.4 Transcribe the following words in both the ARPAbet and the IPA:

a. dark
b. suit
c. greasy
d. wash
e. water

4.5 Write an FST which correctly pronounces strings of dollar amounts like *$45*, *$320*, and *$4100*. If there are multiple ways to pronounce a number you may pick your favorite way.

4.6 Write an FST which correctly pronounces seven-digit phone numbers like *555-1212*, *555-1300*, and so on. You should use a combination of the **paired** and **trailing unit** methods of pronunciation for the last four digits.

4.7 Build an automaton for rule (4.5).

4.8 One difference between one dialect of Canadian English and most dialects of American English is called **Canadian raising**. Bromberger and Halle (1989) note that some Canadian dialects of English raise /aɪ/ to [ʌɪ] and /aʊ/ to [ʌʊ] in stressed position before a voiceless consonant. A simplified version of the rule dealing only with /aɪ/ can be stated as:

CANADIAN RAISING

$$/aɪ/ \rightarrow [ʌɪ] \; / \_\_ \begin{bmatrix} C \\ -voice \end{bmatrix} \tag{4.12}$$

This rule has an interesting interaction with the flapping rule. In some Canadian dialects the word *rider* and *writer* are pronounced differently: *rider* is pronounced [raɪɾɚ] while *writer* is pronounced [rʌɪɾɚ]. Write a two-level rule and an automaton for both the raising rule and the flapping rule which correctly models this distinction. You may make simplifying assumptions as needed.

4.9 Write the lexical entry for the pronunciation of the English past tense (preterite) suffix *-d*, and the two level-rules that express the difference in its pronunciation depending on the previous context. Don't worry about the spelling rules. (Hint: make sure you correctly handle the pronunciation of the past tenses of the words *add*, *pat*, *bake*, and *bag*.)

4.10 Write two-level rules for the Yawelmani Yokuts phenomena of Harmony, Shortening, and Lowering introduced on page 111. Make sure your rules are capable of running in parallel.

4.11 Find 10 stress-neutral name suffixes (look in a phone book) and sketch an FST which would model the pronunciation of names with or without suffixes.

5 PROBABILISTIC MODELS OF PRONUNCIATION AND SPELLING

ALGERNON: *But my own sweet Cecily, I have never written you any letters.*
CECILY: *You need hardly remind me of that, Ernest. I remember only too well that I was forced to write your letters for you. I wrote always three times a week, and sometimes oftener.*
ALGERNON: *Oh, do let me read them, Cecily?*
CECILY: *Oh, I couldn't possibly. They would make you far too conceited. The three you wrote me after I had broken off the engagement are so beautiful, and so badly spelled, that even now I can hardly read them without crying a little.*
Oscar Wilde, *The Importance of being Ernest*

Like Oscar Wilde's fabulous Cecily, a lot of people were thinking about spelling during the last turn of the century. Gilbert and Sullivan provide many examples. *The Gondoliers*' Giuseppe, for example, worries that his private secretary is "shaky in his spelling" while *Iolanthe*'s Phyllis can "spell every word that she uses". Thorstein Veblen's explanation (in his 1899 classic *The Theory of the Leisure Class*) was that a main purpose of the "archaic, cumbrous, and ineffective" English spelling system was to be difficult enough to provide a test of membership in the leisure class. Whatever the social role of spelling, we can certainly agree that many more of us are like Cecily than like Phyllis. Estimates for the frequency of spelling errors in human typed text vary from 0.05% of the words in carefully edited newswire text to 38% in difficult applications like telephone directory lookup (Kukich, 1992).

In this chapter we discuss the problem of detecting and correcting

spelling errors and the very related problem of modeling pronunciation variation for automatic speech recognition and text-to-speech systems. On the surface, the problems of finding spelling errors in text and modeling the variable pronunciation of words in spoken language don't seem to have much in common. But the problems turn out to be isomorphic in an important way: they can both be viewed as problems of *probabilistic transduction*. For speech recognition, given a string of symbols representing the pronunciation of a word in context, we need to figure out the string of symbols representing the lexical or dictionary pronunciation, so we can look the word up in the dictionary. But any given surface pronunciation is ambiguous; it might correspond to different possible words. For example the ARPAbet pronunciation [er] could correspond to reduced forms of the words *her*, *were*, *are*, *their*, or *your*. This ambiguity problem is heightened by **pronunciation variation**; for example the word *the* is sometimes pronounced THEE and sometimes THUH; the word *because* sometimes appears as *because*, sometimes as *'cause*. Some aspects of this variation are systematic; Section 5.7 will survey the important kinds of variation in pronunciation that are important for speech recognition and text-to-speech, and present some preliminary rules describing this variation. High-quality speech synthesis algorithms need to know when to use particular pronunciation variants. Solving both speech tasks requires extending the transduction between surface phones and lexical phones discussed in Chapter 4 with probabilistic variation.

Similarly, given the sequence of letters corresponding to a mis-spelled word, we need to produce an ordered list of possible correct words. For example the sequence *acress* might be a mis-spelling of *actress*, or of *cress*, or of *acres*. We transduce from the "surface" form *acress* to the various possible "lexical" forms, assigning each with a probability; we then select the most probable correct word.

In this chapter we first introduce the problems of detecting and correcting spelling errors, and also summarize typical human spelling error patterns. We then introduce the essential probabilistic architecture that we will use to solve both spelling and pronunciation problems: the **Bayes Rule** and the **noisy channel model**. The Bayes rule and its application to the noisy channel model will play a role in many problems throughout the book, particularly in speech recognition (Chapter 7), part-of-speech tagging (Chapter 8), and probabilistic parsing (Chapter 12).

The Bayes Rule and the noisy channel model provide the probabilistic framework for these problems. But actually solving them requires an algorithm. This chapter introduces an essential algorithm called the **dynamic**

programming algorithm, and various instantiations including the **Viterbi** algorithm, the **minimum edit distance** algorithm, and the **forward** algorithm. We will also see the use of a probabilistic version of the finite-state automaton called the **weighted automaton**.

5.1 Dealing with Spelling Errors

The detection and correction of spelling errors is an integral part of modern word-processors. The very same algorithms are also important in applications in which even the individual letters aren't guaranteed to be accurately identified: **optical character recognition (OCR)** and **on-line handwriting recognition**. **Optical character recognition** is the term used for automatic recognition of machine or hand-printed characters. An optical scanner converts a machine or hand-printed page into a bitmap which is then passed to an OCR algorithm. OCR

On-line handwriting recognition is the recognition of human printed or cursive handwriting as the user is writing. Unlike OCR analysis of handwriting, algorithms for on-line handwriting recognition can take advantage of dynamic information about the input such as the number and order of the strokes, and the speed and direction of each stroke. On-line handwriting recognition is important where keyboards are inappropriate, such as in small computing environments (palm-pilot applications, etc.) or in scripts like Chinese that have large numbers of written symbols, making keyboards cumbersome.

In this chapter we will focus on detection and correction of spelling errors, mainly in typed text, but the algorithms will apply also to OCR and handwriting applications. OCR systems have even higher error rates than human typists, although they tend to make different errors than typists. For example OCR systems often misread "D" as "O" or "ri" as "n", producing 'mis-spelled' words like *dension* for *derision*, or *POQ Bach* for *PDQ Bach*. The reader with further interest in handwriting recognition should consult sources such as Tappert et al. (1990), Hu et al. (1996), and Casey and Lecolinet (1996).

Kukich (1992), in her survey article on spelling correction, breaks the field down into three increasingly broader problems:

1. **non-word error detection:** detecting spelling errors that result in non-words (like *graffe* for *giraffe*)

2. **isolated-word error correction:** correcting spelling errors that result in non-words, for example correcting *graffe* to *giraffe*, but looking only at the word in isolation

3. **context-dependent error detection and correction:** using the context to help detect and correct spelling errors even if they accidentally result in an actual word of English (**real-word errors**). This can happen from typographical errors (insertion, deletion, transposition) which accidently produce a real word (e.g., *there* for *three*), or because the writer substituted the wrong spelling of a homophone or near-homophone (e.g., *dessert* for *desert*, or *piece* for *peace*).

REAL-WORD ERRORS

The next section will discuss the kinds of spelling-error patterns that occur in typed text and OCR and handwriting-recognition input.

5.2 Spelling Error Patterns

The number and nature of spelling errors in human typed text differs from those caused by pattern-recognition devices like OCR and handwriting recognizers. Grudin (1983) found spelling error rates of between 1 and 3% in human typewritten text (this includes both non-word errors and real-word errors). This error rate goes down significantly for copy-edited text. The rate of spelling errors in handwritten text itself is similar; word error rates of between 1.5 and 2.5% have been reported (Kukich, 1992).

The errors of OCR and on-line hand-writing systems vary. Yaeger et al. (1998) propose, based on studies that they warn are inconclusive, that the on-line printed character recognition on Apple Computer's NEWTON MESSAGEPAD had a word accuracy rate of 97–98%, that is, an error rate of 2–3%, but with a high variance (depending on the training of the writer, etc.). It is not clear whether the failure of the NEWTON was because this error rate was optimistic or because a 2–3% error rate is unacceptable. More recent devices, like 3Com's Palm Pilot, often use a special input script (like the Palm Pilot's "Graffiti") instead of allowing arbitrary handwriting. OCR error rates also vary widely depending on the quality of the input; (Lopresti and Zhou, 1997) suggest that OCR letter-error rates typically range from 0.2% for clean, first-generation copy to 20% or worse for multigeneration photocopies and faxes.

In an early study, Damerau (1964) found that 80% of all misspelled words (non-word errors) in a sample of human keypunched text were caused by **single-error misspellings**: a single one of the following errors:[1]

- **insertion**: mistyping *the* as *ther* INSERTION
- **deletion**: mistyping *the* as *th* DELETION
- **substitution**: mistyping *the* as *thw* SUBSTITUTION
- **transposition**: mistyping *the* as *hte* TRANSPOSITION

Because of this study, much following research has focused on the correction of single-error misspellings. Indeed, the first algorithm we will present later in this chapter relies on the large proportion of single-error misspellings.

Kukich (1992) breaks down human typing errors into two classes. **Typographic errors** (for example misspelling *spell* as *speel*), are generally related to the keyboard. **Cognitive errors** (for example misspelling *separate* as *seperate*) are caused by writers who don't know how to spell the word. Grudin (1983) found that the keyboard was the strongest influence on the errors produced; typographic errors constituted the majority of all error types. For example consider substitution errors, which were the most common error type for novice typists, and the second most common error type for expert typists. Grudin found that immediately adjacent keys in the same row accounted for 59% of the novice substitutions and 31% of the error substitutions (e.g., *smsll* for *small*). Adding in errors in the same column and **homologous** errors (hitting the corresponding key on the opposite side of the keyboard with the other hand), a total of 83% of the novice substitutions and 51% of the expert substitutions could be considered keyboard-based errors. Cognitive errors included phonetic errors (substituting a phonetically equivalent sequence of letters (*seperate* for *separate*) and homonym errors (substituting *piece* for *peace*). Homonym errors will be discussed in Chapter 7 when we discuss real-word error correction.

While typing errors are usually characterized as substitutions, insertions, deletions, or transpositions, OCR errors are usually grouped into five classes: substitutions, multisubstitutions, space deletions or insertions, and

[1] In another corpus, Peterson (1986) found that single-error misspellings accounted for an even higher percentage of all misspelled words (93–95%). The difference between the 80% and the higher figure may be due to the fact that Damerau's text included errors caused in transcription to punched card forms, errors in keypunching, and errors caused by paper tape equipment (!) in addition to purely human misspellings.

failures. Lopresti and Zhou (1997) give the following example of common OCR errors:

Correct:
The quick brown fox jumps over the lazy dog.
Recognized:
'lhe q˜ ick brown foxjurnps ovcr tb l azy dog.

Substitutions ($e \rightarrow c$) are generally caused by visual similarity (rather than keyboard distance), as are multisubstitutions ($T \rightarrow$ *'l*, $m \rightarrow$ *rn*, *he* $\rightarrow$ *b*). Multisubstitutions are also often called **framing errors**. Failures (represented by the tilde character '˜ ': $u \rightarrow$ ˜) are cases where the OCR algorithm does not select any letter with sufficient accuracy.

5.3 Detecting Non-Word Errors

Detecting non-word errors in text, whether typed by humans or scanned, is most commonly done by the use of a dictionary. For example, the word *foxjurnps* in the OCR example above would not occur in a dictionary. Some early research (Peterson, 1986) had suggested that such spelling dictionaries would need to be kept small, because large dictionaries contain very rare words that resemble misspellings of other words. For example *wont* is a legitimate but rare word but is a common misspelling of *won't*. Similarly, *veery* (a kind of thrush) might also be a misspelling of *very*. Based on a simple model of single-error misspellings, Peterson showed that it was possible that 10% of such misspellings might be "hidden" by real words in a 50,000 word dictionary, but that 15% of single-error misspellings might be "hidden" in a 350,000-word dictionary. In practice, Damerau and Mays (1989) found that this was not the case; while some misspellings were hidden by real words in a larger dictionary, in practice the larger dictionary proved more help than harm.

Because of the need to represent productive inflection (the *-s* and *ed* suffixes) and derivation, dictionaries for spelling error detection usually include models of morphology, just as the dictionaries for text-to-speech we saw in Chapters 3 and 4. Early spelling error detectors simply allowed any word to have any suffix – thus Unix SPELL accepts bizarre prefixed words like *misclam* and *antiundoggingly* and suffixed words based on *the* like *thehood* and *theness*. Modern spelling error detectors use more linguistically-motivated morphological representations (see Chapter 3).

5.4 PROBABILISTIC MODELS

This section introduces probabilistic models of pronunciation and spelling variation. These models, particularly the **Bayesian inference** or **noisy channel** model, will be applied throughout this book to many different problems.

We claimed earlier that the problem of ASR pronunciation modeling, and the problem of spelling correction for typing or for OCR, can be modeled as problems of mapping from one string of symbols to another. For speech recognition, given a string of symbols representing the pronunciation of a word in context, we need to figure out the string of symbols representing the lexical or dictionary pronunciation, so we can look the word up in the dictionary. Similarly, given the incorrect sequence of letters in a mis-spelled word, we need to figure out the correct sequence of letters in the correctly spelled word.

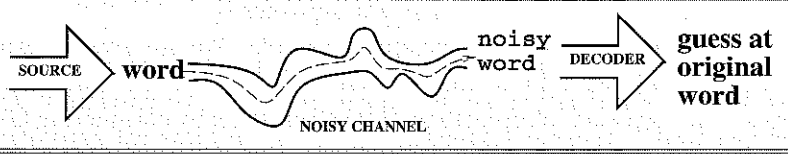


Figure 5.1 The noisy channel model.

NOISY CHANNEL

The intuition of the **noisy channel** model (see Figure 5.1) is to treat the surface form (the "reduced"" pronunciation or misspelled word) as an instance of the lexical form (the "lexical" pronunciation or correctly-spelled word) which has been passed through a noisy communication channel. This channel introduces "noise" which makes it hard to recognize the "true" word. Our goal is then to build a model of the channel so that we can figure out how it modified this "true" word and hence recover it. For the complete speech recognition tasks, there are many sources of "noise"; variation in pronunciation, variation in the realization of phones, acoustic variation due to the channel (microphones, telephone networks, etc.). Since this chapter focuses on pronunciation, what we mean by "noise" here is the variation in pronunciation that masks the lexical or "canonical" pronunciation; the other sources of noise in a speech recognition system will be discussed in Chapter 7. For spelling error detection, what we mean by noise is the spelling errors which mask the correct spelling of the word. The metaphor of the noisy channel comes from the application of the model to speech recognition in the IBM labs in the 1970s (Jelinek, 1976). But the algorithm itself is a special case

BAYESIAN

of **Bayesian inference** and as such has been known since the work of Bayes (1763). Bayesian inference or Bayesian classification was applied successfully to language problems as early as the late 1950s, including the OCR work of Bledsoe in 1959, and the seminal work of Mosteller and Wallace (1964) on applying Bayesian inference to determine the authorship of the Federalist papers.

In Bayesian classification, as in any classification task, we are given some observation and our job is to determine which of a set of classes it belongs to. For speech recognition, imagine for the moment that the observation is the string of phones which make up a word as we hear it. For spelling error detection, the observation might be the string of letters that constitute a possibly-misspelled word. In both cases, we want to classify the observations into words; thus in the speech case, no matter which of the many possible ways the word *about* is pronounced (see Chapter 4) we want to classify it as *about*. In the spelling case, no matter how the word *separate* is misspelled, we'd like to recognize it as *separate*.

Let's begin with the pronunciation example. We are given a string of phones (say [ni]). We want to know which word corresponds to this string of phones. The Bayesian interpretation of this task starts by considering all possible classes—in this case, all possible words. Out of this universe of words, we want to chose the word which is most probable given the observation we have ([ni]). In other words, we want, out of all words in the vocabulary V the single word such that $P(\text{word}|\text{observation})$ is highest. We use $\hat{\mathbf{w}}$ to mean "our estimate of the correct w", and we'll use $\mathbf{O}$ to mean "the observation sequence [ni]" (we call it a sequence because we think of each letter as an individual observation). Then the equation for picking the best word given is:

V
$\hat{w}$
o

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(w|O) \tag{5.1}$$

The function $\operatorname{argmax}_x f(x)$ means "the x such that $f(x)$ is maximized". While (5.1) is guaranteed to give us the optimal word w, it is not clear how to make the equation operational; that is, for a given word w and observation sequence O we don't know how to directly compute $P(w|O)$. The intuition of Bayesian classification is to use Bayes' rule to transform (5.1) into a product of two probabilities, each of which turns out to be easier to compute than $P(w|O)$. Bayes' rule is presented in (5.2); it gives us a way to break down $P(x|O)$ into three other probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{5.2}$$

We can see this by substituting (5.2) into (5.1) to get (5.3):

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} \frac{P(O|w)P(w)}{P(O)} \tag{5.3}$$

The probabilities on the right-hand side of (5.3) are for the most part easier to compute than the probability $P(w|O)$ that we were originally trying to maximize in (5.1). For example, $P(w)$, the probability of the word itself, we can estimate by the frequency of the word. And we will see below that $P(O|w)$ turns out to be easy to estimate as well. But $P(O)$, the probability of the observation sequence, turns out to be harder to estimate. Luckily, we can ignore $P(O)$. Why? Since we are maximizing over all words, we will be computing $\frac{P(O|w)P(w)}{P(O)}$ for each word. But $P(O)$ doesn't change for each word; we are always asking about the most likely word string for the same observation O, which must have the same probability $P(O)$. Thus:

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} \frac{P(O|w)P(w)}{P(O)} = \underset{w \in V}{\operatorname{argmax}} P(O|w)\,P(w) \tag{5.4}$$

To summarize, the most probable word w given some observation O can be computing by taking the product of two probabilities for each word, and choosing the word for which this product is greatest. These two terms have names; $P(w)$ is called the **Prior probability**, and $P(O|w)$ is called the **likelihood**.

PRIOR

LIKELIHOOD

Key Concept #3.

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} \overbrace{P(O|w)}^{\text{likelihood}} \overbrace{P(w)}^{\text{prior}} \tag{5.5}$$

In the next sections we will show how to compute these two probabilities for the probabilities of pronunciation and spelling.

5.5 Applying the Bayesian Method to Spelling

There are many algorithms for spelling correction; we will focus on the Bayesian (or noisy channel) algorithm because of its generality. Chapter 6 will show how this algorithm can be extended to model real-word spelling errors; this section will focus on non-word spelling errors. The noisy channel approach to spelling correction was first suggested by Kernighan et al. (1990); their program, `correct`, takes words rejected by the Unix `spell` program, generates a list of potential correct words, rank them according to Equation (5.5), and picks the highest-ranked one.

Let's walk through the algorithm as it applies to Kernighan et al.'s (1990) example misspelling *acress*. The algorithm has two stages: *proposing candidate corrections* and *scoring the candidates*.

In order to propose candidate corrections Kernighan et al. make the simplifying assumption that the correct word will differ from the misspelling by a single insertion, deletion, substitution, or transposition. As Damerau's (1964) results show, even though this assumption causes the algorithm to miss some corrections, it should handle most spelling errors in human typed text. The list of candidate words is generated from the typo by applying any single transformation which results in a word in a large on-line dictionary. Applying all possible transformations to *acress* yields the list of candidate words in Figure 5.2.

		Transformation			
Error	Correction	Correct Letter	Error Letter	Position (Letter #)	Type
acress	actress	t	–	2	deletion
acress	cress	–	a	0	insertion
acress	caress	ca	ac	0	transposition
acress	access	c	r	2	substitution
acress	across	o	e	3	substitution
acress	acres	–	2	5	insertion
acress	acres	–	2	4	insertion

Figure 5.2 Candidate corrections for the misspelling *acress*, together with the transformations that would have produced the error (after Kernighan et al. (1990)). "–" represents a null letter.

The second stage of the algorithm scores each correction by Equation 5.4. Let t represent the typo (the misspelled word), and let c range over the set C of candidate corrections. The most likely correction is then:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(t|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \tag{5.6}$$

As in Equation (5.4) we have omitted the denominator in Equation (5.6) since the typo t, and hence its probability $P(t)$, is constant for all c. The prior probability of each correction $P(c)$ can be estimated by counting how often the word c occurs in some corpus, and then **normalizing** these counts by the

NORMALIZING

total count of all words.[2] So the probability of a particular correction word c is computed by dividing the count of c by the number N of words in the corpus. Zero counts can cause problems, and so we will add .5 to all the counts. This is called "smoothing", and will be discussed in Chapter 6; note that in Equation (5.7) we can't just divide by the total number of words N since we added .5 to the counts of all the words, so we add .5 for each of the V words in the vocabulary).

$$P(c) = \frac{C(c) + 0.5}{N + 0.5V} \tag{5.7}$$

Chapter 6 will talk more about the role of corpora in computing prior probabilities; for now let's use the corpus of Kernighan et al. (1990), which is the 1988 AP newswire corpus of 44 million words. Thus N is 44 million. Since in this corpus the word *actress* occurs 1343 times, the word *acres* 2879 times, and so on, the resulting prior probabilities are as follows:

c	**freq(c)**	**p(c)**
actress	1343	.0000315
cress	0	.000000014
caress	4	.0000001
access	2280	.000058
across	8436	.00019
acres	2879	.000065

Computing the likelihood term $p(t|c)$ exactly is an unsolved (unsolveable?) research problem; the exact probability that a word will be mistyped depends on who the typist was, how familiar they were with the keyboard they were using, whether one hand happened to be more tired than the other, etc. Luckily, while $p(t|c)$ cannot be computed exactly, it can be *estimated* pretty well, because the most important factors predicting an insertion, deletion, transposition are simple local factors like the identity of the correct letter itself, how the letter was misspelled, and the surrounding context. For example, the letters m and n are often substituted for each other; this is partly a fact about their identity (these two letters are pronounced similarly and they are next to each other on the keyboard), and partly a fact about context (because they are pronounced similarly, they occur in similar contexts).

One simple way to estimate these probabilities is the one that Kernighan et al. (1990) used. They ignored most of the possible influences on the probability of an error and just estimated e.g. $p(acress|across)$ using

[2] Normalizing means dividing by some total count so that the resulting probabilities fall legally between 0 and 1.

CONFUSION MATRIX

the number of times that e was substituted for o in some large corpus of errors. This is represented by a **confusion matrix**, a square 26×26 table which represents the number of times one letter was incorrectly used instead of another. For example, the cell labeled $[o,e]$ in a substitution confusion matrix would give the count of times that e was substituted for o. The cell labeled $[t,s]$ in an insertion confusion matrix would give the count of times that t was inserted after s. A confusion matrix can be computed by hand-coding a collection of spelling errors with the correct spelling and then counting the number of times different errors occurred (this has been done by Grudin (1983)). Kernighan et al. (1990) used four confusion matrices, one for each type of single-error:

- del$[x,y]$ contains the number of times in the training set that the characters xy in the correct word were typed as x.
- ins$[x,y]$ contains the number of times in the training set that the character x in the correct word was typed as xy.
- sub$[x,y]$ the number of times that x was typed as y.
- trans$[x,y]$ the number of times that xy was typed as yx.

Note that they chose to condition their insertion and deletion probabilities on the previous character; they could also have chosen to condition on the following character. Using these matrices, they estimated $p(t|c)$ as follows (where c_p is the pth character of the word c):

$$P(t|c) = \begin{cases} \frac{\text{del}[c_{p-1},c_p]}{\text{count}[c_{p-1}c_p]}, \text{ if deletion} \\ \frac{\text{ins}[c_{p-1},t_p]}{\text{count}[c_{p-1}]}, \text{ if insertion} \\ \frac{\text{sub}[t_p,c_p]}{\text{count}[c_p]}, \text{ if substitution} \\ \frac{\text{trans}[c_p,c_{p+1}]}{\text{count}[c_pc_{p+1}]}, \text{ if transposition} \end{cases} \tag{5.8}$$

Figure 5.3 shows the final probabilities for each of the potential corrections; the prior (from Equation (5.7)) is multiplied by the likelihood (computed using Equation (5.8) and the confusion matrices). The final column shows the "normalized percentage".

This implementation of the Bayesian algorithm predicts *acres* as the correct word (at a total normalized percentage of 45%), and *actress* as the second most likely word. Unfortunately, the algorithm was wrong here: The writer's intention becomes clear from the context: ... *was called a "stellar and versatile* **acress** *whose combination of sass and glamour has defined her...*". The surrounding words make it clear that *actress* and not *acres* was

c	freq(c)	p(c)	p(t|c)	p(t|c)p(c)	%
actress	1343	.0000315	.000117	3.69×10^{-9}	**37%**
cress	0	.000000014	.00000144	2.02×10^{-14}	**0%**
caress	4	.0000001	.00000164	1.64×10^{-13}	**0%**
access	2280	.000058	.000000209	1.21×10^{-11}	**0%**
across	8436	.00019	.0000093	1.77×10^{-9}	**18%**
acres	2879	.000065	.0000321	2.09×10^{-9}	**21%**
acres	2879	.000065	.0000342	2.22×10^{-9}	**23%**

Figure 5.3 Computation of the ranking for each candidate correction. Note that the highest ranked word is not *actress* but *acres* (the two lines at the bottom of the table), since *acres* can be generated in two ways. The *del*[], *ins*[], *sub*[], and *trans*[] confusion matrices are given in full in Kernighan et al. (1990).

the intended word; Chapter 6 will show how to augment the computation of the prior probability to use the surrounding words.

The algorithm as we have described it requires hand-annotated data to train the confusion matrices. An alternative approach used by Kernighan et al. (1990) is to compute the matrices by iteratively using this very spelling error correction algorithm itself. The iterative algorithm first initializes the matrices with equal values; thus any character is equally likely to be deleted, equally likely to be substituted for any other character, etc. Next the spelling error correction algorithm is run on a set of spelling errors. Given the set of typos paired with their corrections, the confusion matrices can now be recomputed, the spelling algorithm run again, and so on. This clever method turns out to be an instance of the important **EM** algorithm (Dempster et al., 1977) that we will discuss in Chapter 7 and Appendix D. Kernighan et al. (1990)'s algorithm was evaluated by taking some spelling errors that had two potential corrections, and asking three human judges to pick the best correction. Their program agreed with the majority vote of the human judges 87% of the time.

5.6 Minimum Edit Distance

The previous section showed that the Bayesian algorithm, as implemented with confusion matrices, was able to rank candidate corrections. But Kernighan et al. (1990) relied on the simplifying assumption that each word had only a single spelling error. Suppose we wanted a more powerful algorithm

which could handle the case of multiple errors? We could think of such an algorithm as a general solution to the problem of **string distance**. The "string distance" is some metric of how alike two strings are to each other. The Bayesian method can be viewed as a way of applying such an algorithm to the spelling error correction problem; we pick the candidate word which is "closest" to the error in the sense of having the highest probability given the error.

DISTANCE

One of the most popular classes of algorithms for finding string distance are those that use some version of the **minimum edit distance** algorithm, named by Wagner and Fischer (1974) but independently discovered by many people; see the History section. The minimum edit distance between two strings is the minimum number of editing operations (insertion, deletion, substitution) needed to transform one string into another. For example the gap between intention and execution is five operations, which can be represented in three ways; as a **trace**, an **alignment**, or a **operation list** as show in Figure 5.4.

MINIMUM EDIT DISTANCE

ALIGNMENT

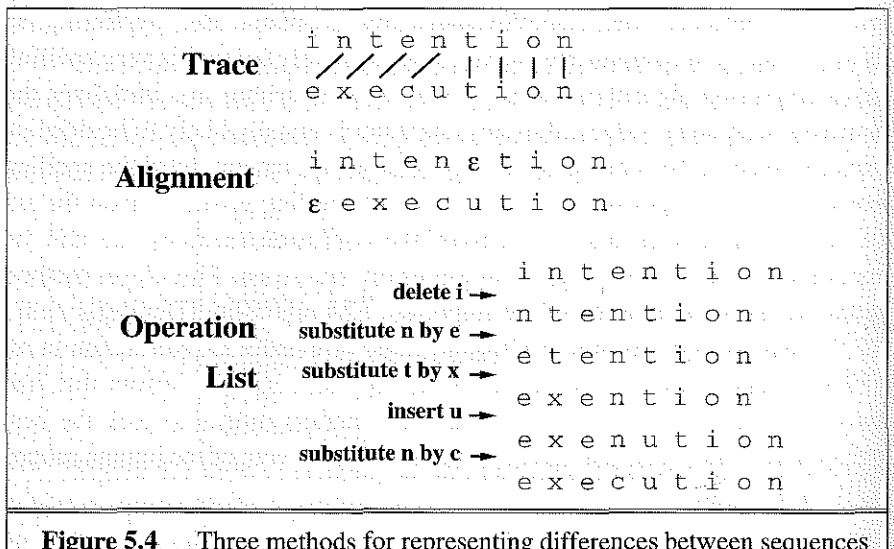


Figure 5.4 Three methods for representing differences between sequences (after Kruskal (1983))

We can also assign a particular cost or weight to each of these operations. The **Levenshtein** distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966). Thus the Levenshtein distance between *intention* and *execution* is 5. Levenshtein also proposed an alternate version of his metric

in which each insertion or deletion has a cost of one, and substitutions are not allowed (equivalent to allowing substitution, but giving each substitution a cost of 2, since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between *intention* and *execution* is 8. We can also weight operations by more complex functions, for example by using the confusion matrices discussed above to assign a probability to each operation. In this case instead of talking about the "minimum edit distance" between two strings, we are talking about the "maximum probability **alignment**" of one string with another. If we do this, an augmented minimum edit distance algorithm which multiplies the probabilities of each transformation can be used to estimate the Bayesian likelihood of a multiple-error typo given a candidate correction.

DYNAMIC PROGRAMMING

The minimum edit distance is computed by **dynamic programming**. Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to subproblems. This class of algorithms includes the most commonly-used algorithms in speech and language processing, among them the **minimum edit distance** algorithm for spelling error correction the **Viterbi** algorithm and the **forward** algorithm which are used both in speech recognition and in machine translation, and the **CYK** and **Earley** algorithm used in parsing. We will introduce the minimum-edit-distance, Viterbi, and forward algorithms in this chapter and Chapter 7, the Earley algorithm in Chapter 10, and the CYK algorithm in Chapter 12.

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems. For example, consider the sequence or "path" of transformed words that comprise the minimum edit distance between the strings *intention* and *execution*. Imagine some string (perhaps it is *exention*) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if *exention* is in the optimal operation-list, then the optimal sequence must also include the optimal path from *intention* to *exention*. Why? If there were a shorter path from *intention* to *exention* then we could use it instead, resulting in a shorter overall path, and the optimal sequence wouldn't be optimal, thus leading to a contradiction.

Dynamic programming algorithms for sequence comparison work by creating a distance matrix with one column for each symbol in the target sequence and one row for each symbol in the source sequence (i.e., target along the bottom, source along the side). For minimum edit distance, this matrix is the *edit-distance* matrix. Each cell *edit-distance*$[i,j]$ contains the distance

between the first i characters of the target and the first j characters of the source. Each cell can be computed as a simple function of the surrounding cells; thus starting from the beginning of the matrix it is possible to fill in every entry. The value in each cell is computing by taking the minimum of the three possible paths through the matrix which arrive there:

$$P(t|c) = \min \begin{cases} distance[i-1,j] + ins\text{-}cost(target_i) \\ distance[i-1,j-1] + subst\text{-}cost(source_j, target_i) \\ distance[i,j-1] + del\text{-}cost(source_j)) \end{cases} \quad (5.9)$$

The algorithm itself is summarized in Figure 5.5, while Figure 5.6 shows the results of applying the algorithm to the distance between *intention* and *execution* assuming the version of Levenshtein distance in which insertions and deletions each have a cost of 1 and substitutions have a cost of 2.

function MIN-EDIT-DISTANCE(*target, source*) **returns** *min-distance*

$n \leftarrow$ LENGTH(*target*)
$m \leftarrow$ LENGTH(*source*)
Create a distance matrix *distance[n+1,m+1]*
distance[0,0] $\leftarrow 0$
for each column i **from** 0 **to** n **do**
  for each row j **from** 0 **to** m **do**
    distance[i, j] $\leftarrow$ MIN(*distance*[$i-1,j$] + *ins-cost*(*target*$_i$),
      distance[$i-1,j-1$] + *subst-cost*(*source*$_j$, target$_i$),
      distance[$i,j-1$] + *del-cost*(*source*$_j$))

Figure 5.5 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms.

5.7 ENGLISH PRONUNCIATION VARIATION

> When any of the fugitives of Ephraim said: 'Let me go over,' the men of Gilead said unto him: 'Art thou an Ephraimite?' If he said: 'Nay'; then said they unto him: 'Say now Shibboleth'; and he said 'Sibboleth'; for he could not frame to pronounce it right; then they laid hold on him, and slew him at the fords of the Jordan.
>
> Judges 12:5-6

n	9	10	11	10	11	12	11	10	9	**8**
o	8	9	10	9	10	11	10	9	**8**	9
i	7	8	9	8	9	10	9	**8**	9	10
t	6	7	8	7	8	9	**8**	9	10	11
n	5	6	7	6	7	**8**	9	10	11	12
e	4	5	6	**5**	**6**	7	8	9	10	11
t	3	4	**5**	6	7	8	9	10	11	12
n	2	3	**4**	5	6	7	8	8	10	11
i	1	**2**	3	4	5	6	7	8	9	10
#	**0**	1	2	3	4	5	6	7	8	9
	#	e	x	e	c	u	t	i	o	n

Figure 5.6 Computation of minimum edit distance between *intention* and *execution* via algorithm of Figure 5.5, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. Substitution of a character for itself has a cost of 0.

This passage from Judges is a rather gory reminder of the political importance of pronunciation variation. Even in our (hopefully less political) computational applications of pronunciation, it is important to correctly model how pronunciations can vary. We have already seen that a phoneme can be realized as different allophones in different phonetic environments. We have also shown how to write rules and transducers to model these changes for speech synthesis. Unfortunately, these models significantly simplified the nature of pronunciation variation. In particular, pronunciation variation is caused by many factors in addition to the phonetic environment. This section summarizes some of these kinds of variation; the following section will introduce the probabilistic tools for modeling it.

Pronunciation variation is extremely widespread. Figure 5.7 shows the most common pronunciations of the words *because* and *about* from the hand-transcribed Switchboard corpus of American English telephone conversations. Note the wide variation in pronunciation for these two words when spoken as part of a continuous stream of speech.

What causes this variation? There are two broad classes of pronunciation variation: **lexical variation** and **allophonic variation**. We can think of lexical variation as a difference in what segments are used to represent the word in the lexicon, while allophonic variation is a difference in how the individual segments change their value in different contexts. In Figure 5.7, most of the variation in pronunciation is allophonic; that is, due to the influ-

LEXICAL VARIATION

ALLOPHONIC VARIATION

because			about		
IPA	**ARPAbet**	**%**	**IPA**	**ARPAbet**	**%**
[bikʌz]	[b iy k ah z]	27%	[əbaʊ]	[ax b aw]	32%
[bɨkʌz]	[b ix k ah z]	14%	[əbaʊt]	[ax b aw t]	16%
[kʌz]	[k ah z]	7%	[baʊ]	[b aw]	9%
[kəz]	[k ax z]	5%	[ʌbaʊ]	[ix b aw]	8%
[bɨkəz]	[b ix k ax z]	4%	[ɨbaʊt]	[ix b aw t]	5%
[bɪkʌz]	[b ih k ah z]	3%	[ɨbæ]	[ix b ae]	4%
[bəkʌz]	[b ax k ah z]	3%	[əbær]	[ax b ae dx]	3%
[kʊz]	[k uh z]	2%	[baʊr]	[b aw dx]	3%
[ks]	[k s]	2%	[bæ]	[b ae]	3%
[kɨz]	[k ix z]	2%	[baʊt]	[b aw t]	3%
[kɪz]	[k ih z]	2%	[əbaʊr]	[ax b aw dx]	3%
[bikʌʒ]	[b iy k ah zh]	2%	[əbæ]	[ax b ae]	3%
[bikʌs]	[b iy k ah s]	2%	[bɑ]	[b aa]	3%
[bikʌ]	[b iy k ah]	2%	[bær]	[b ae dx]	3%
[bikɑz]	[b iy k aa z]	2%	[ɨbaʊr]	[ix b aw dx]	2%
[əz]	[ax z]	2%	[ɨbɑt]	[ix b aa t]	2%

Figure 5.7 The 16 most common pronunciations of *because* and *about* from the hand-transcribed Switchboard corpus of American English conversational telephone speech (Godfrey et al., 1992; Greenberg et al., 1996).

ence of the surrounding sounds, syllable structure, and so forth. But the fact that the word *because* can be pronounced either as monosyllabic *'cause* or bisyllabic *because* is probably a lexical fact, having to do perhaps with the level of informality of speech.

SOCIOLINGUISTIC

An important source of lexical variation (although it can also affect allophonic variation) is **sociolinguistic** variation. Sociolinguistic variation is due to extralinguistic factors such as the social identity or background of the speaker. One kind of sociolinguistic variation is **dialect variation**. Speakers of some deep-southern dialects of American English use a monophthong or near-monophthong [a] or [aɛ] instead of a diphthong in some words with the vowel [aɪ]. In these dialects *rice* is pronounced [raːs]. African-American Vernacular English (AAVE) has many of the same vowel differences from General American as does Southern American English, and also has individual words with specific pronunciations such as [bɪdnɪs] for *business* and [æks] for *ask*.

DIALECT VARIATION

For older speakers or those not from the American West or Midwest, the words *caught* and *cot* have different vowels ([kɔt] and [kɑt]

respectively). Young American speakers or those from the West pronounce the two words *cot* and *caught* the same; the vowels [ɔ] and [ɑ] are usually not distinguished in these dialects. For some speakers from New York City like the first author's parents, the words *Mary* ([mɛɪri]), *marry* ([mæri]), and *merry* ([mɛri]) are all pronounced differently, while other New York City speakers like the second author pronounce *Mary*, and *merry* identically, but differently than *marry*. Most American speakers pronounce all three of these words identically as ([mɛri]). Students who are interested in dialects of English should consult Wells (1982), the most comprehensive study of dialects of English around the world.

Other sociolinguistic differences are due to **register** or **style** rather than dialect. In a pronunciation difference that is due to style, the same speaker might pronounce the same word differently depending on who they were talking to or what the social situation is; this is probably the case when choosing between *because* and *'cause* above. One of the most well-studied examples of style-variation is the suffix *-ing* (as in *something*), which can be pronounced [ɪŋ] or /ɪn/ (this is often written *somethin'*). Most speakers use both forms; as Labov (1966) shows, they use [ɪŋ] when they are being more formal, and [ɪn] when more casual. In fact whether a speaker will use [ɪŋ] or [ɪn] in a given situation varies markedly according to the social context, the gender of the speaker, the gender of the other speaker, and so on. Wald and Shopen (1981) found that men are more likely to use the non-standard form [ɪn] than women, that both men and women are more likely to use more of the standard form [ɪŋ] when the addressee is a women, and that men (but not women) tend to switch to [ɪn] when they are talking with friends.

REGISTER
STYLE

Where lexical variation happens at the lexical level, allophonic variation happens at the surface form and reflects phonetic and articulatory factors.[3] For example, most of the variation in the word *about* in Figure 5.7 was caused by changes in one of the two vowels or by changes to the final [t]. Some of this variation is due to the allophonic rules we have already discussed for the realization of the phoneme /t/. For example the pronunciation of *about* as [əbaʊɾ]/[ax b aw dx]) has a flap at the end because the next word was the word *it*, which begins with a vowel; the sequence *about it* was pronounced [əbaʊɾɨ]/[ax b aw dx ix]). Similarly, note that final [t] is often deleted; (*about* as [baʊ]/[b aw]). Considering these cases as "deleted" is actually a simplification; many of these "deleted" cases of [t] are actually

[3] For some purposes we distinguish between allophonic variation and what are called "optional phonological rules"; for the purposes of this textbook we will lump these both together as "allophonic variation".

realized as a slight change to the vowel quality called **glottalization** which are not represented in these transcriptions.

When we discussed these rules earlier, we implied that they were deterministic; given an environment, a rule always applies. This is by no means the case. Each of these allophonic rules is dependent on a complicated set of factors that must be interpreted probabilistically. In the rest of this section we summarize more of these rules and talk about the influencing factors.

COARTICULATION

Many of these rules model **coarticulation**, which is a change in a segment due to the movement of the articulators in neighboring segments. Most allophonic rules relating English phoneme to their allophones can be grouped into a small number of types: assimilation, dissimilation, deletion, flapping, vowel reduction, and epenthesis.

ASSIMILATION

Assimilation is the change in a segment to make it more like a neighboring segment. The dentalization of [t] to ([t̪]) before the dental consonant [θ] is an example of assimilation. Another common type of assimilation

PALATALIZATION

in English and cross-linguistically is **palatalization**. Palatalization occurs when the constriction for a segment occurs closer to the palate than it normally would, because the following segment is palatal or alveolo-palatal. In the most common cases, /s/ becomes [ʃ], /z/ becomes [ʒ], /t/ becomes [tʃ and /d/ becomes dʒ]. We saw one case of palatalization in Figure 5.7 in the pronunciation of *because* as [bikʌʒ] (ARPAbet [b iy k ah zh]). Here the final segment of *because*, a lexical /z/, is realized as [ʒ], because the following word was *you've*. So the sequence *because you've* was pronounced [bikʌʒuv]. A simple version of a palatalization rule might be expressed as follows; Figure 5.8 shows examples from the Switchboard corpus.

$$\left\{ \begin{matrix} [s] \\ [z] \\ [t] \\ [d] \end{matrix} \right\} \Rightarrow \left\{ \begin{matrix} [ʃ] \\ [ʒ] \\ [tʃ] \\ [dʒ] \end{matrix} \right\} / \_\_ \{ y \} \tag{5.10}$$

Note in Figure 5.8 that whether a [t] is palatalized depends on lexical factors like word frequency ([t] is more likely to be palatalized in frequent words and phrases).

DELETION

Deletion is quite common in English speech. We saw examples of deletion of final /t/ above, in the words *about* and *it*. /t/ and /d/ are often deleted before consonants, or when they are part of a sequence of two or three consonants; Figure 5.9 shows some examples.

$$\left\{ \begin{matrix} t \\ d \end{matrix} \right\} \Rightarrow \emptyset / \mathrm{V} \_\_ \mathrm{C} \tag{5.11}$$

The many factors that influence the deletion of /t/ and /d/ have been extensively studied. For example /d/ is more likely to be deleted than /t/.

Phrase	IPA Lexical	IPA Reduced	ARPAbet Reduced
set your	[sɛtjɔr]	[sɛtʃɚ]	[s eh ch er]
not yet	[nɑtjɛt]	[nɑtʃɛt]	[n aa ch eh t]
last year	[læstjir]	[læstʃir]	[l ae s ch iy r]
what you	[wʌtju]	[wətʃu]	[w ax ch uw]
this year	[ðɪsjir]	[ðɪʃir]	[dh ih sh iy r]
because you've	[bikʌzjuv]	[bikʌʒuv]	[b iy k ah zh uw v]
did you	[dɪdju]	[dɪdʒyʌ]	[d ih jh y ah]

Figure 5.8 Examples of palatalization from the Switchboard corpus; the lemma *you* (including *your*, *you've*, and *you'd*) was by far the most common cause of palatalization, followed by *year(s)* (especially in the phrases *this year* and *last year*).

Phrase	IPA Lexical	IPA Reduced	ARPAbet Reduced
find him	[faɪndhɪm]	[faɪnɪm]	[f ay n ix m]
around this	[ɚraʊndðɪs]	[ɨraʊnɪs]	[ix r aw n ih s]
mind boggling	[maɪnbɔglɪŋ]	[maɪnbɔglɪŋ]	[m ay n b ao g el ih ng]
most places	[moʊstpleɪsɨz]	[moʊspleɪsɨz]	[m ow s p l ey s ix z]
draft the	[dræftði]	[dræfði]	[d r ae f dh iy]
left me	[lɛftmi]	[lɛfmi]	[l eh f m iy]

Figure 5.9 Examples of /t/ and /d/ deletion from Switchboard. Some of these examples may have glottalization instead of being completely deleted.

Both are more likely to be deleted before a consonant (Labov, 1972). The final /t/ and /d/ in the words *and* and *just* are particularly likely to be deleted (Labov, 1975; Neu, 1980). Wolfram (1969) found that deletion is more likely in faster or more casual speech, and that younger people and males are more likely to delete. Deletion is more likely when the two words surrounding the segment act as a sort of phrasal unit, either occurring together frequently (Bybee, 1996), having a high **mutual information** or **trigram predictability** (Gregory et al., 1999), or being tightly connected for other reasons (Zwicky, 1972). Fasold (1972), Labov (1972), and many others have shown that deletion is less likely if the word-final /t/ or /d/ is the past tense ending. For example in Switchboard, deletion is more likely in the word *around* (73% /d/-deletion) than in the word *turned* (30% /d/-deletion) even though the two words have similar frequencies.

The **flapping** rule is significantly more complicated than we suggested in Chapter 4, as a number of scholars have pointed out (see especially Rhodes (1992)). The preceding vowel is highly likely to be stressed, although this is not necessary (for example there is commonly a flap in the word *thermometer* [θɚˈmɑmɨɾɚ]). The following vowel is highly likely to be unstressed, although again this is not necessary. /t/ is much more likely to flap than /d/. There are complicated interactions with syllable, foot, and word boundaries. Flapping is more likely to happen when the speaker is speaking more quickly, and is more likely to happen at the end of a word when it forms a collocation (high mutual information) with the following word (Gregory et al., 1999). Flapping is less likely to happen when a speaker **hyperarticulates**, i.e. uses a particularly clear form of speech, which often happens when users are talking to computer speech recognition systems (Oviatt et al., 1998). There is a nasal flap [ɾ̃] whose tongue movements resemble the oral flap but in which the velum is lowered. Finally, flapping doesn't always happen, even when the environment is appropriate; thus the flapping rule, or transducer, needs to be probabilistic, as we will see below.

HYPERARTICULATES

We have saved for last one of the most important phonological processes: **vowel reduction**, in which many vowels in unstressed syllables are realized as **reduced vowels**, the most common of which is **schwa** ([ə]). Stressed syllables are those in which more air is pushed out of the lungs; stressed syllables are longer, louder, and usually higher in pitch than unstressed syllables. Vowels in unstressed syllables in English often don't have their full form; the articulatory gesture isn't as complete as for a full vowel. As a result the shape of the mouth is somewhat neutral; the tongue is neither particularly high nor particularly low. For example the second vowels in *parakeet* is schwa: [pærəkit].

REDUCED VOWELS

SCHWA

While schwa is the most common reduced vowel, it is not the only one, at least not in some dialects. Bolinger (1981) proposed three reduced vowels: a reduced mid vowel [ə], a reduced front vowel [ɨ], and a reduced rounded vowel [ɵ]. But the majority of computational pronunciation lexicons or computational models of phonology systems limit themselves to one reduced vowel ([ə]) (for example PRONLEX and CELEX) or at most two ([ə] =ARPABET [ax] and [i] = ARPAbet [ix]). Miller (1998) was able to train a neural net to automatically categorize a vowel as [ə] or [ɨ] based only on the phonetic context, which suggests that for speech recognition and text-to-speech purposes, one reduced vowel is probably adequate. Indeed, Wells (1982, p. 167–168) notes that [ə] and [ɨ] are falling together in many dialects of English including General American and Irish, among others, a

phenomenon he calls **weak vowel merger**.

A final note: not all unstressed vowels are reduced; any vowel, and diphthongs in particular can retain their full quality even in unstressed position. For example the vowel [eɪ] (ARPAbet [ey]) can appear in stressed position as in the word *eight*) ['eɪt] or unstressed position as in the word *always* ['ɔ.weɪz]. Whether a vowel is reduced depends on many factors. For example the word *the* can be pronounced with a full vowel ði or reduced vowel ðə. It is more likely to be pronounced with the reduced vowel ðə in fast speech, in more casual situations, and when the following word begins with a consonant. It is more likely to be pronounced with the full vowel ði when the following word begins with a vowel or when the speaker is having "planning problems"; speakers are more likely to use a full vowel than a reduced one if they don't know what they are going to say next (Fox Tree and Clark, 1997). See Keating et al. (1994) and Jurafsky et al. (1998) for more details on factors effecting vowel reduction in the TIMIT and Switchboard corpora. Other factors influencing reduction include the frequency of the word, whether this is the final vowel in a phrase, and even the idiosyncracies of individual speakers.

5.8 The Bayesian Method for Pronunciation

HEAD KNIGHT OF NI:	Ni!
KNIGHTS OF NI:	Ni! Ni! Ni! Ni! Ni!
ARTHUR:	Who are you?
HEAD KNIGHT:	We are the Knights Who Say... 'Ni'!
RANDOM:	Ni!
ARTHUR:	No! Not the Knights Who Say 'Ni'!
HEAD KNIGHT:	The same!
BEDEVERE:	Who are they?
HEAD KNIGHT:	We are the keepers of the sacred words: 'Ni', 'Peng', and 'Neee–wom'!

Graham Chapman, John Cleese, Eric Idle, Terry Gilliam, Terry Jones, and Michael Palin, *Monty Python and the Holy Grail* 1975.

The Bayesian algorithm that we used to pick the optimal correction for a spelling error can be used to solve what is often called the **pronunciation** subproblem in speech recognition. In this task, we are given a series of phones and our job is to compute the most probable word which generated them. For this chapter, we will simplify the problem in an important way by assuming the correct string of phones. A real speech recognizer relies on

probabilistic estimators for each phone, so it is never sure about the identity of any phone. We will relax this assumption in Chapter 7; for now, let's look at the simpler problem.

We'll also begin with another simplification by assuming that we already know where the word boundaries are. Later in the chapter, we'll show that we can simultaneously find word boundaries ("segment") and model pronunciation variation.

Consider the particular problem of interpreting the sequence of phones [ni], when it occurs after the word *I* at the beginning of a sentence. Stop and see if you can think of any words which are likely to have been pronounced [ni] before you read on. The word "Ni" is not allowed.

You probably thought of the word *knee*. This word is in fact pronounced [ni]. But an investigation of the Switchboard corpus produces a total of 7 words which can be pronounced [ni]! The seven words are *the*, *neat*, *need*, *new*, *knee*, *to*, and *you*.

How can the word *the* be pronounced [ni]? The explanation for this pronunciation (and all the others except the one for *knee*) lies in the contextually-induced pronunciation variation we discussed in Chapter 4. For example, we saw that [t] and [d] were often deleted word finally, especially before coronals; thus the pronunciation of *neat* as [ni] happened before the word *little* (*neat little* → [nilə|]). The pronunciation of *the* as [ni] is caused by the regressive assimilation process also discussed in Chapter 4. Recall that in nasal assimilation, phones before or after nasals take on nasal manner of articulation. Thus [θ] can be realized as [n]. The many cases of *the* pronounced as [ni] in Switchboard occurred after words like *in*, *on*, and *been* (so *in the* → [ɪnni]). The pronunciation of *new* as [ni] occurred most frequently in the word *New York*; the vowel [u] has fronted to [i] before a [y].

The pronunciation of *to* as [ni] occurred after the work *talking* (*talking to you* → [tɔkɪniyu]); here the [u] is palatalized by the following [y] and the [n] is functioning jointly as the final sound of *talking* and the initial sound of *to*. Because this phone is part of two separate words we will not try to model this particular mapping; for the rest of this section let's consider only the following five words as candidate lexical forms for [ni]: *knee*, *the*, *neat*, *need*, *new*.

We saw in the previous section that the Bayesian spelling error correction algorithm had two components: candidate generation, and candidate scoring. Speech recognizers often use an alternative architecture, trading off speech for storage. In this architecture, each pronunciation is expanded in advance with all possible variants, which are then pre-stored with their

scores. Thus there is no need for candidate generation; the word [ni] is simply stored with the list of words that can generate it. Let's assume this method and see how the prior and likelihood are computed for each word.

We will be choosing the word whose product of prior and likelihood is the highest, according to Equation (5.12), where y represents the sequence of phones (in this case [ni] and w represents the candidate word [*the*, *new*, etc.]). The most likely word is then:

$$\hat{w} = \underset{w \in W}{\operatorname{argmax}} \overbrace{P(y|w)}^{\text{likelihood}} \overbrace{P(w)}^{\text{prior}} \tag{5.12}$$

We could choose to generate the likelihoods $p(y|w)$ by using a set of confusion matrices as we did for spelling error correction. But it turns out that confusion matrices don't do as well for pronunciation as for spelling. While misspelling tends to change the form of a word only slightly, the changes in pronunciation between a lexical and surface form are much greater. Confusion matrices only work well for single-errors, which, as we saw above, are common in misspelling. Furthermore, recall from Chapter 4 that pronunciation variation is strongly affected by the surrounding phones, lexical frequency, and stress and other prosodic factors. Thus probabilistic models of pronunciation variation include a lot more factors than a simple confusion matrix can include.

One simple way to generate pronunciation likelihoods is via **probabilistic rules**. Probabilistic rules were first proposed for pronunciation by (Labov, 1969) (who called them **variable rules**). The idea is to take the rules of pronunciation variation we saw in Chapter 4 and associate them with probabilities. We can then run these probabilistic rules over the lexicon and generate different possible surface forms each with its own probability. For example, consider a simple version of a nasal assimilation rule which explains why *the* can be pronounced [ni]; a word-initial [ð] becomes [n] if the preceding word ended in [n] or sometimes [m]:

PROBABILISTIC RULES

$$[.15]\ \text{ð} \Rightarrow n \ /\ [+nasal]\ \#\_\_ \tag{5.13}$$

The [.15] to the left of the rule is the probability; this can be computed from a large-enough labeled corpus such as the transcribed portion of Switchboard. Let *ncount* be the number of times lexical [ð] is realized word-initially by surface [n] when the previous word ends in a nasal (91 in the Switchboard corpus). Let *envcount* be the total number of times lexical [ð] occurs (whatever its surface realization) when the previous word ends in a nasal (617 in the Switchboard corpus). The resulting probability is:

$$P(\text{ð} \rightarrow n \;/\; [+nasal]\ \#\_\_) = \frac{ncount}{envcount} = \frac{91}{617} = .15$$

We can build similar probabilistic versions of the assimilation and deletion rules which account for the [ni] pronunciation of the other words. Figure 5.10 shows sample rules and the probabilities trained on the Switchboard pronunciation database.

Word	Rule Name	Rule	P
the	nasal assimilation	ð ⇒ n / [+*nasal*] #__	[.15]
neat	final t deletion	$t \Rightarrow \emptyset \;/\; V$ __ #	[.52]
need	final d deletion	$d \Rightarrow \emptyset \;/\; V$ __ #	[.11]
new	u fronting	$u \Rightarrow i \;/$ __ #[y]	[.36]

Figure 5.10 Simple rules of pronunciation variation due to context in continuous speech accounting for the pronunciation of each of these words as [ni].

We now need to compute the prior probability $P(w)$ for each word. For spelling correction we did this by using the relative frequency of the word in a large corpus; a word which occurred 44,000 times in 44 million words receives the probability estimate $\frac{44,000}{44,000,000}$ or .001. For the pronunciation problem, let's take our prior probabilities from a collection of a written and a spoken corpus. The Brown Corpus is a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.) which was assembled at Brown University in 1963–1964 (Kučera and Francis, 1967; Francis, 1979; Francis and Kučera, 1982). The Switchboard Treebank corpus is a 1.4 million word collection of telephone conversations. Together they let us sample from both the written and spoken genres. The table below shows the probabilities for our five words; each probability is computed from the raw frequencies by normalizing by the number of words in the combined corpus (plus .5 * the number of word types; so the total denominator is 2,486,075 + 30,836):

w	freq(w)	p(w)
knee	61	.000024
the	114,834	.046
neat	338	.00013
need	1417	.00056
new	2625	.001

Now we are almost ready to answer our original question: what is the most likely word given the pronunciation [ni] and given that the previous word was *I* at the beginning of a sentence. Let's start by multiplying together our estimates for $p(w)$ and $p(y|w)$ to get an estimate; we show them sorted from most probable to least probable (*the* has a probability of 0 since the previous phone was not [n], and hence there is no other rule allowing [ð] to be realized as [n]):

Word	p(y|w)	p(w)	p(y|w)p(w)
new	.36	.001	.00036
neat	.52	.00013	.000068
need	.11	.00056	.000062
knee	1.00	.000024	.000024
the	0	.046	0

Our algorithm suggests that *new* is the most likely underlying word. But this is the wrong answer; the string [ni] following the word *I* came in fact from the word *need* in the Switchboard corpus. One way that people are able to correctly solve this task is word-level knowledge; people know that the word string *I need* ... is much more likely than the word string *I new* .... We don't need to abandon our Bayesian model to handle this fact; we just need to modify it so that our model also knows that *I need* is more likely than *I new*. In Chapter 6 we will see that we can do this by using a slightly more intelligent estimate of $p(w)$ called a **bigram** estimate; essentially we consider the probability of *need* following *I* instead of just the individual probability of *need*.

This Bayesian algorithm is in fact part of all modern speech recognizers. Where the algorithms differ strongly is how they detect individual phones in the acoustic signal, and on which search algorithm they use to efficiently compute the Bayesian probabilities to find the proper string of words in connected speech (as we will see in Chapter 7).

Decision Tree Models of Pronunciation Variation

In the previous section we saw how hand-written rules could be augmented with probabilities to model pronunciation variation. Riley (1991) and Withgott and Chen (1993) suggested an alternative to writing rules by hand, which has proved quite useful: automatically inducing lexical-to-surface pronunciations mappings from a labeled corpus with a **decision tree**, particularly with the kind of decision tree called a **Classification and Regression Tree** (**CART**) (Breiman et al., 1984). A decision tree takes a situation described by a set of features and classifies it into a category and an associated probability. For pronunciation, a decision tree can be trained to take a lexical phone and various contextual features (surrounding phones, stress and syllable structure information, perhaps lexical identity) and select an appropriate surface phone to realize it. We can think of the confusion matrices we used in spelling error correction above as degenerate decision trees; thus the substitution matrix takes a lexical phone and outputs a probability distribution over potential surface phones to be substituted. The advantage of decision trees is that they can be automatically induced from a labeled corpus, and that they are concise: Decision trees pick out only the relevant features and thus suffer less from sparseness than a matrix, which has to condition on every neighboring phone.

DECISION TREE

CART

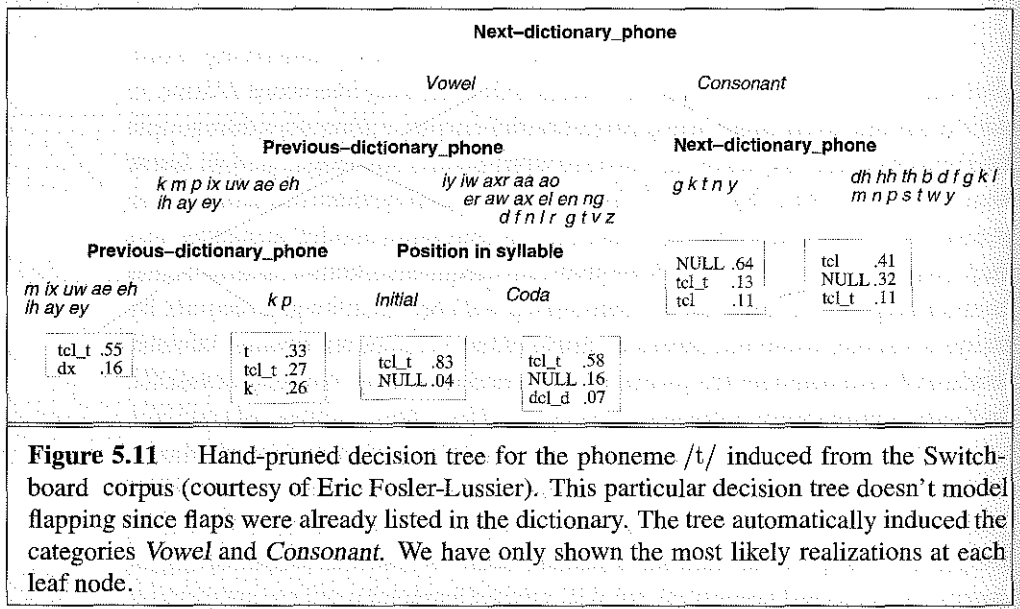


Figure 5.11 Hand-pruned decision tree for the phoneme /t/ induced from the Switchboard corpus (courtesy of Eric Fosler-Lussier). This particular decision tree doesn't model flapping since flaps were already listed in the dictionary. The tree automatically induced the categories *Vowel* and *Consonant*. We have only shown the most likely realizations at each leaf node.

For example, Figure 5.11 shows a decision tree for the pronunciation of the phoneme /t/ induced from the Switchboard corpus. While this tree doesn't including flapping (there is a separate tree for flapping) it does model the fact that /t/ is more likely to be deleted before a consonant than before a vowel. Note, in fact, that the tree automatically induced the classes *Vowel* and *Consonant*. Furthermore note that if /t/ is not deleted before a consonant, it is likely to be unreleased. Finally, notice that /t/ is very unlikely to be deleted in syllable onset position.

Readers with interest in decision tree modeling of pronunciation should consult Riley (1991), Withgott and Chen (1993), and a textbook with an introduction to decision trees such as Russell and Norvig (1995).

5.9 Weighted Automata

We said earlier that for purposes of efficiency a lexicon is often stored with the most likely kinds of pronunciation variation pre-compiled. The two most common representation for such a lexicon are the **trie** and the **weighted finite-state automaton/transducer** (or **probabilistic FSA/FST**) (Pereira et al., 1994). We will leave the discussion of the trie to Chapter 7, and concentrate here on the weighted automaton. WEIGHTED

The weighted automaton is a simple augmentation of the finite automaton in which each arc is associated with a probability, indicating how likely that path is to be taken. The probability on all the arcs leaving a node must sum to 1. Figure 5.12 shows two weighted automata for the word *tomato*, adapted from Russell and Norvig (1995). The top automaton shows two possible pronunciations, representing the dialect difference in the second vowel. The bottom one shows more pronunciations (how many?) representing optional reduction or deletion of the first vowel and optional flapping of the final [t].

A **Markov chain** is a special case of a weighted automaton in which the input sequence uniquely determines which states the automaton will go through. Because they can't represent inherently ambiguous problems, a Markov chain is only useful for assigning probabilities to unambiguous sequences; thus the N-gram models to be discussed in Chapter 6 are Markov chains since each word is treated as if it was unambiguous. In fact the weighted automata used in speech and language processing can be shown to be equivalent to **Hidden Markov Models (HMMs)**. Why do we introduce weighted automata in this chapter and HMMs in Chapter 7? The MARKOV CHAIN

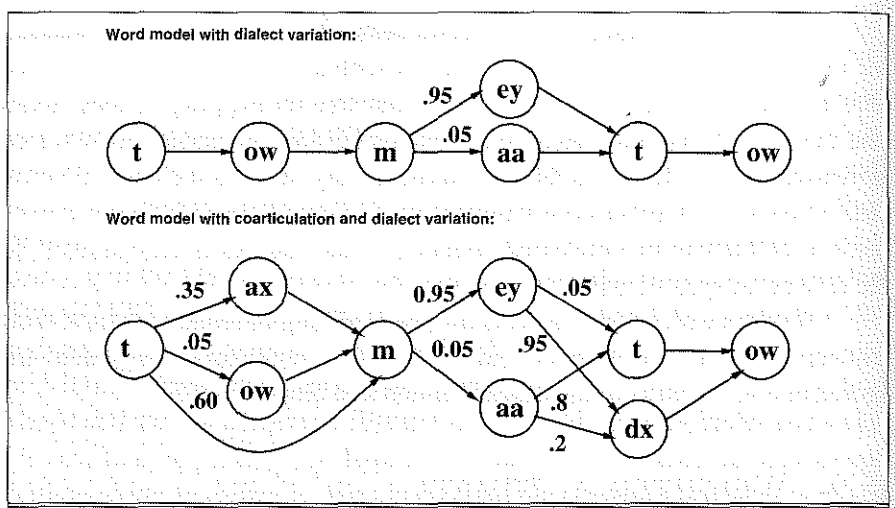


Figure 5.12 You say [t ow m ey t ow] and I say [t ow m aa t ow]. Two pronunciation networks for the word *tomato*, adapted from Russell and Norvig (1995). The top one models sociolinguistic variation (some British or eastern American dialects); the bottom one adds in coarticulatory effects. Note the correlation between allophonic and sociolinguistic variation; the dialect with the vowel [ey] is more likely to flap than the other dialect.

two models offer a different metaphor; it is sometimes easier to think about certain problems as weighted-automata than as HMMs. The weighted automaton metaphor is often applied when the input alphabet maps relatively neatly to the underlying alphabet. For example, in the problem of correcting spelling errors in typewritten input, the input sequence consists of letters and the states of the automaton can correspond to letters. Thus it is natural to think of the problem as transducing from a set of symbols to the same set of symbols with some modifications, and hence weighted automata are naturally used for spelling error correction. In the problem of correcting errors in hand-written input, the input sequence is visual, and the input alphabet is an alphabet of lines and angles and curves. Here instead of transducing from an alphabet to itself, we need to do classification on some input sequence before considering it as a sequence of states. Hidden Markov Models provide a more appropriate metaphor, since they naturally handle separate alphabets for input sequences and state sequences. But since any probabilistic automaton in which the input sequence does not uniquely specify the state sequence can be modeled as an HMM, the difference is one of metaphor rather than explanatory power.

Weighted automata can be created in many ways. One way, first proposed by Cohen (1989) is to start with on-line pronunciation dictionaries and use hand-written rules of the kind we saw above to create different potential surface forms. The probabilities can then be assigned either by counting the number of times each pronunciation occurs in a corpus, or if the corpus is too sparse, by learning probabilities for each rule and multiplying out the rule probabilities for each surface form (Tajchman et al., 1995). Finally these weighted rules, or alternatively the decision trees we discussed in the last section, can be automatically compiled into a weighted finite-state transducer (Sproat and Riley, 1996). Alternatively, for very common words, we can simply find enough examples of the pronunciation in a transcribed corpus to build the model by just combining all the pronunciations into a network (Wooters and Stolcke, 1994).

The networks for *tomato* above were shown merely as illustration and are not from any real system; Figure 5.13 shows an automaton for the word *about* which is trained on actual pronunciations from the Switchboard corpus (we discussed these pronunciations in Chapter 4).

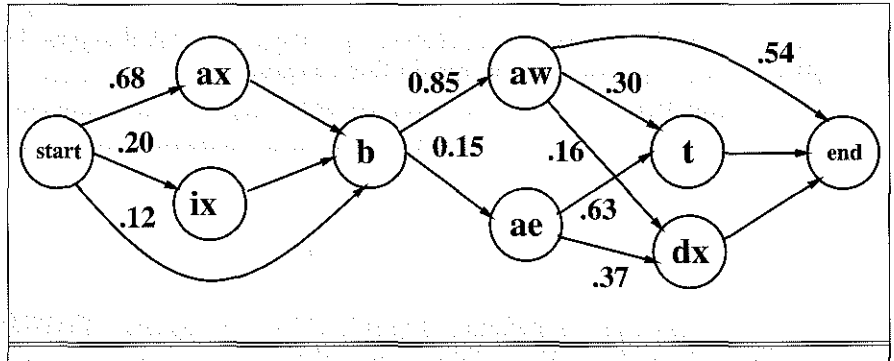


Figure 5.13 A pronunciation network for the word *about*, from the actual pronunciations in the Switchboard corpus.

Computing Likelihoods from Weighted Automata: The Forward Algorithm

One advantage of an automaton-based lexicon is that there are efficient algorithms for generating the probabilities that are needed to implement the Bayesian method of correct-word-identification of Section 5.8. These algorithms apply to weighted automata and also to the **Hidden Markov Models** that we will discuss in Chapter 7. Recall that in our example the Bayesian

method is given as input a series of phones [n iy], and must choose between the words *the*, *neat*, *need*, *new*, and *knee*. This was done by computing two probabilities: the prior probability of each word, and the likelihood of the phone string [n iy] given each word. When we discussed this example earlier, we said that for example the likelihood of [n iy] given the word *need* was .11, since we computed a probability of .11 for the *final-d-deletion* rule from our Switchboard corpus. This probability is transparent for *need* since there were only two possible pronunciations ([n iy] and [n iy d]). But for words like *about*, visualizing the different probabilities is more complex. Using a precompiled weighted automata can make it simpler to see all the different probabilities of different paths through the automaton.

There is a very simple algorithm for computing the likelihood of a string of phones given the weighted automaton for a word. This algorithm,
FORWARD the **forward** algorithm, is an essential part of ASR systems, although in this chapter we will only be working with a simple usage of the algorithm. This is because the forward algorithm is particularly useful when there are multiple paths through an automaton which can account for the input; this is not the case in the weighted automata in this chapter, but will be true for the HMMS of Chapter 7. The forward algorithm is also an important step in defining the **Viterbi** algorithm that we will see later in this chapter.

Let's begin by giving a formal definition of a weighted automaton and of the input and output to the likelihood computation problem. A weighted automaton consists of

1. a sequence of states $q = (q_0 q_1 q_2 \ldots q_n)$, each corresponding to a phone, and
2. a set of transition probabilities between states, a_{01}, a_{12}, a_{13}, encoding the probability of one phone following another.

We represent the states as nodes, and the transition probabilities as edges between nodes; an edge exists between two nodes if there is a non-zero transition probability between the two nodes.[4] The sequences of symbols

[4] We have used two "special" states (often called **non-emitting states**) as the start and end state; it is also possible to avoid the use of these states. In that case, an automaton must specify two more things:

1. π, an initial probability distribution over states, such that π_i is the probability that the automaton will start in state i. Of course, some states j may have $\pi_j = 0$, meaning that they cannot be initial states.
2. a set of legal accepting states.

that are input to the model (if we are thinking of it as recognizer) or which are produced by the model (if we are thinking of it as a generator) are generally called the **observation sequence**, referred to as $O = (o_1 o_2 o_3 \ldots o_t)$. (Upper-case letters are used for a sequence and lower-case letters for an individual element of a sequence). We will use this terminology when talking about weighted automata and later when talking about HMMs.

OBSERVATION SEQUENCE

Figure 5.14 shows an automaton for the word *need* with a sample observation sequence.

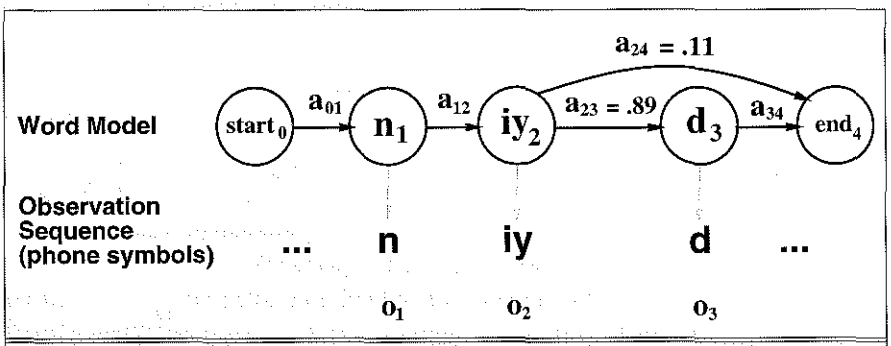


Figure 5.14 A simple weighted automaton or Markov chain pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. The transition probabilities a_{xy} between two states x and y are 1.0 unless otherwise specified.

This task of determining which underlying word might have produced an observation sequence is called the **decoding** problem. Recall that in order to find which of the candidate words was most probable given the observation sequence [n iy], we need to compute the product $P(O|w)P(w)$ for each candidate word (*the*, *need*, *neat*, *knee*, *new*), i.e. the likelihood of the observation sequence O given the word w times the prior probability of the word.

DECODING

The forward algorithm can be run to perform this computation for each word; we give it an observation sequence and the pronunciation automaton for a word and it will return $P(O|w)P(w)$. Thus one way to solve the decoding problem is to run the forward algorithm separately on each word and choose the word with the highest value. As we saw earlier, the Bayesian method produces the wrong result for pronunciation [n iy] as part of the word sequence *I need* (its first choice is the word *new*, and the second choice is *neat*; *need* is only the third choice). Since the forward algorithm is just a way of implementing the Bayesian approach, it will return the exact same

rankings. (We will see in Chapter 6 how to augment the algorithm with **bigram** probabilities which will enable it to make use of the knowledge that the previous word was *I*).

The forward algorithm takes as input a pronunciation network for each candidate word. Because the word *the* only has the pronunciation [n iy] after nasals, and since we are assuming the actual context of this word was after the word *I* (no nasal), we will skip that word and look only at *new*, *neat*, *need*, and *knee*. Note in Figure 5.15 that we have augmented each network with the probability of each word, computed from the frequency that we saw on page 167.

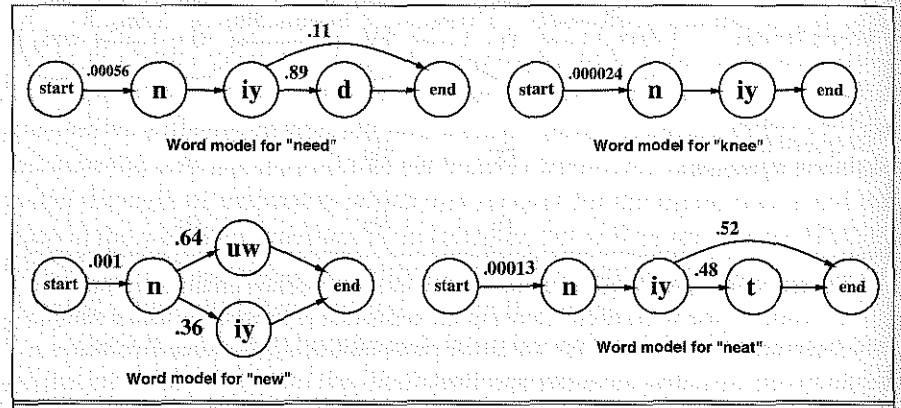


Figure 5.15 Pronunciation networks for the words *need*, *neat*, *new*, and *knee*. All networks are simplified from the actual pronunciations in the Switchboard corpus. Each network has been augmented by the unigram probability of the word (i.e., its normalized frequency from the Switchboard+Brown corpus). Word probabilities are not usually included as part of the pronunciation network for a word; they are added here to simplify the exposition of the forward algorithm.

The forward algorithm is another **dynamic programming** algorithm, and can be thought of as a slight generalization of the minimum edit distance algorithm. Like the minimum edit distance algorithm, it uses a table to store intermediate values as it builds up the probability of the observation sequence. Unlike the minimum edit distance algorithm, the rows are labeled not just by states which always occur in linear order, but implicitly by a *state-graph* which has many ways of getting from one state to another. In the minimum edit distance algorithm, we filled in the matrix by just computing the value of each cell from the three cells around it. With the forward

algorithm, on the other hand, a state might be entered by any other state, and so the recurrence relation is somewhat more complicated. Furthermore, the forward algorithm computes the *sum* of the probabilities of all possible paths that could generate the observation sequence, where the minimum edit distance computed the *minimum* such probability.[5] Each cell of the forward algorithm matrix, $forward[t, j]$ represents the probability of being in state j after seeing the first t observations, given the automaton λ. Since we have augmented our graphs with the word probability $p(w)$, our example of the forward algorithm here is actually computing this likelihood times $p(w)$. The value of each cell $forward[t, j]$ is computed by summing over the probabilities of every path that could lead us to this cell. Formally, each cell expresses the following probability:

$$forward[t, j] = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)\ P(w) \tag{5.14}$$

Here $q_t = j$ means "the probability that the tth state in the sequence of states is state j". We compute this probability by summing over the extensions of all the paths that lead to the current cell. An extension of a path from a state i at time $t-1$ is computed by multiplying the following three factors:

1. the **previous path probability** from the previous cell forward$[t-1, i]$,
2. the **transition probability** a_{ij} from previous state i to current state j, and
3. the **observation likelihood** b_{jt} that current state j matches observation symbol t. For the weighted automata that we consider here, b_{jt} is 1 if the observation symbol matches the state, and 0 otherwise. Chapter 7 will consider more complex observation likelihoods.

The algorithm is described in Figure 5.16.

Figure 5.17 shows the forward algorithm applied to the word *need*. The algorithm applies similarly to the other words which can produce the string [n iy], resulting in the probabilities on page 167. In order to compute the most probable underlying word, we run the forward algorithm separately on each of the candidate words, and choose the one with the highest probability. Chapter 7 will give further details of the mathematics of the forward algorithm and introduce the related forward-backward algorithm.

[5] The forward algorithm computes the *sum* because there may be multiple paths through the network which explain a given observation sequence. Chapter 7 will take up this point in more detail.

```
function FORWARD(observations,state-graph) returns forward-probability

  num-states ← NUM-OF-STATES(state-graph)
  num-obs ← length(observations)
  Create probability matrix forward[num-states + 2, num-obs + 2]
  forward[0,0] ← 1.0
  for each time step t from 0 to num-obs do
    for each state s from 0 to num-states do
      for each transition s' from s specified by state-graph
        forward[s',t+1] ← forward[s,t] * a[s, s'] * b[s', o_t]
  return the sum of the probabilities in the final column of forward
```

Figure 5.16 The forward algorithm for computing likelihood of observation sequence given a word model. $a[s,s']$ is the transition probability from current state s to next state s', and $b[s',o_t]$ is the observation likelihood of s' given o_t. For the weighted automata that we consider here, $b[s',o_t]$ is 1 if the observation symbol matches the state, and 0 otherwise.

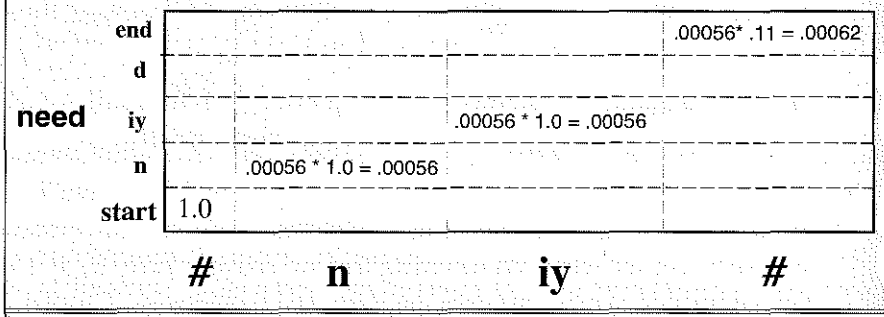


Figure 5.17 The forward algorithm applied to the word *need*, computing the probability $P(O|w)P(w)$. While this example doesn't require the full power of the forward algorithm, we will see its use on more complex examples in Chapter 7.

Decoding: The Viterbi Algorithm

The forward algorithm as we presented it seems a bit of an overkill. Since only one path through the pronunciation networks will match the input string, why use such a big matrix and consider so many possible paths? Furthermore, as a decoding method, it seems rather inefficient to run the forward algorithm once for each word (imagine how inefficient this would be if we were computing likelihoods for all possible sentences rather than all possible

words!) Part of the reason that the forward algorithm seems like overkill is that we have immensely simplified the pronunciation problem by assuming that our input consists of sequences of unambiguous symbols. We will see in Chapter 7 that when the observation sequence is a set of noisy acoustic values, there are many possibly paths through the automaton, and the forward algorithm will play an important role in summing these paths.

But it is true that having to run it separately on each word makes the forward algorithm a very inefficient decoding method. Luckily, there is a simple variation on the forward algorithm called the **Viterbi** algorithm which allows us to consider all the words simultaneously and still compute the most likely path. The term **Viterbi** is common in speech and language processing, but like the forward algorithm this is really a standard application of the classic **dynamic programming** algorithm, and again looks a lot like the **minimum edit distance** algorithm. The Viterbi algorithm was first applied to speech recognition by Vintsyuk (1968), but has what Kruskal (1983) calls a 'remarkable history of multiple independent discovery and publication'; see the History section at the end of the chapter for more details. The name Viterbi is the one which is most commonly used in speech recognition, although the terms **DP alignment** (for **Dynamic Programming alignment**), **dynamic time warping** and **one-pass decoding** are also commonly used. The term is applied to the decoding algorithm for weighted automata and Hidden Markov Models on a single word and also to its more complex application to continuous speech, as we will see in Chapter 7. In this chapter we will show how the algorithm is used to find the best path through a network composed of single words, as a result choosing the word which is most probable given the observation sequence string of words.

VITERBI

DYNAMIC TIME WARPING

The version of the Viterbi algorithm that we will present takes as input a single weighted automaton and a set of observed phones $o = (o_1 o_2 o_3 \dots o_t)$ and returns the most probable state sequence $q = (q_1 q_2 q_3 \dots q_t)$, together with its probability. We can create a single weighted automaton by combining the pronunciation networks for the four words in parallel with a single start and a single end state. Figure 5.18 shows the combined network.

Figure 5.19 shows pseudocode for the Viterbi algorithm. Like the minimum edit distance and forward algorithm, the Viterbi algorithm sets up a probability matrix, with one column for each time index t and one row for each state in the state graph. Also like the forward algorithm, each column has a cell for each state q_i in the single combined automaton for the four words. In fact, the code for the Viterbi algorithm should look exactly like the code for the forward algorithm with two modifications. First, where

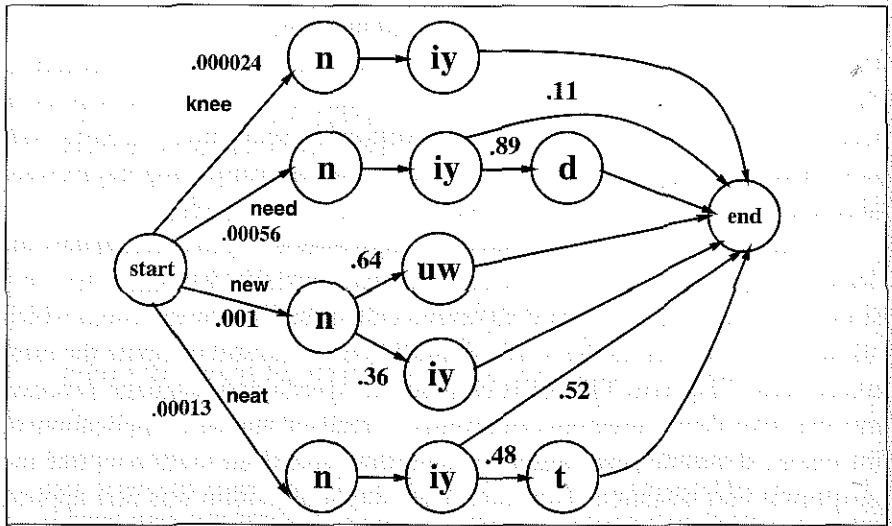


Figure 5.18 The pronunciation networks for the words *need*, *neat*, *new*, and *knee* combined into a single weighted automaton. Again, word probabilities are not usually considered part of the pronunciation network for a word; they are added here to simplify the exposition of the Viterbi algorithm.

the forward algorithm places the *sum* of all previous paths into the current cell, the Viterbi algorithm puts the *max* of the previous paths into the current cell.

The algorithm first creates $N+2$ or four state columns. The first column is an initial pseudo-observation, the second corresponds to the first observation phone [n], the third to [iy] and the fourth to a final pseudo-observation. We begin in the first column by setting the probability of the *start* state to 1.0, and the other probabilities to 0; the reader should find this in Figure 5.20. Cells with probability 0 are simply left blank for readability.

Then we move to the next state; as with the forward algorithm, for every state in column 0, we compute the probability of moving into each state in column 1. The value $viterbi[t, j]$ is computed by taking the maximum over the extensions of all the paths that lead to the current cell. An extension of a path from a state i at time $t-1$ is computed by multiplying the same three factors we used for the forward algorithm:

1. the **previous path probability** from the previous cell $forward[t-1, i]$,
2. the **transition probability** a_{ij} from previous state i to current state j, and
3. the **observation likelihood** b_{jt} that current state j matches observation symbol t. For the weighted automata that we consider here, b_{jt} is 1 if

function VITERBI(*observations* of len *T*,*state-graph*) **returns** *best-path*

num-states ← NUM-OF-STATES(*state-graph*)
Create a path probability matrix *viterbi[num-states+2,T+2]*
viterbi[0,0] ← 1.0
for each time step *t* **from** 0 **to** *T* **do**
  for each state *s* **from** 0 **to** *num-states* **do**
    for each transition s' from *s* specified by *state-graph*
      new-score ← *viterbi*[*s*, *t*] * $a[s,s']$ * $b_{s'}(o_t)$
      if ((*viterbi*[s',*t+1*] = 0) || (*new-score* > *viterbi*[s', *t+1*]))
        then
          viterbi[s', *t+1*] ← *new-score*
          back-pointer[s', *t+1*] ← *s*
Backtrace from highest probability state in the final column of *viterbi[]* and return path

Figure 5.19 Viterbi algorithm for finding optimal sequence of states in continuous speech recognition, simplified by using phones as inputs. Given an observation sequence of phones and a weighted automaton (state graph), the algorithm returns the path through the automaton which has maximum probability and accepts the observation sequence. $a[s,s']$ is the transition probability from current state s to next state s', and $b[s',o_t]$ is the observation likelihood of s' given o_t. For the weighted automata that we consider here, $b[s',o_t]$ is 1 if the observation symbol matches the state, and 0 otherwise.

the observation symbol matches the state, and 0 otherwise. Chapter 7 will consider more complex observation likelihoods.

In Figure 5.20, in the column for the input *n*, each word starts with [n], and so each has a non-zero probability in the cell for the state *n*. Other cells in that column have zero entries, since their states don't match n. When we proceed to the next column, each cell that matches iy gets updated with the contents of the previous cell times the transition probability to that cell. Thus the value of $viterbi[2,\text{iy}_{new}]$ for the iy state of the word *new* is the product of the "word" probability of *new* times the probability of *new* being pronounced with the vowel *iy*. Notice that if we look only at this iy column, that the word *need* is currently the "most-probable" word. But when we move to the final column, the word *new* will win out, since *need* has a smaller transition probability to *end* (.11) than *new* does (1.0). We can now follow the backpointers and backtrace to find the path that gave us this final probability of .00036.

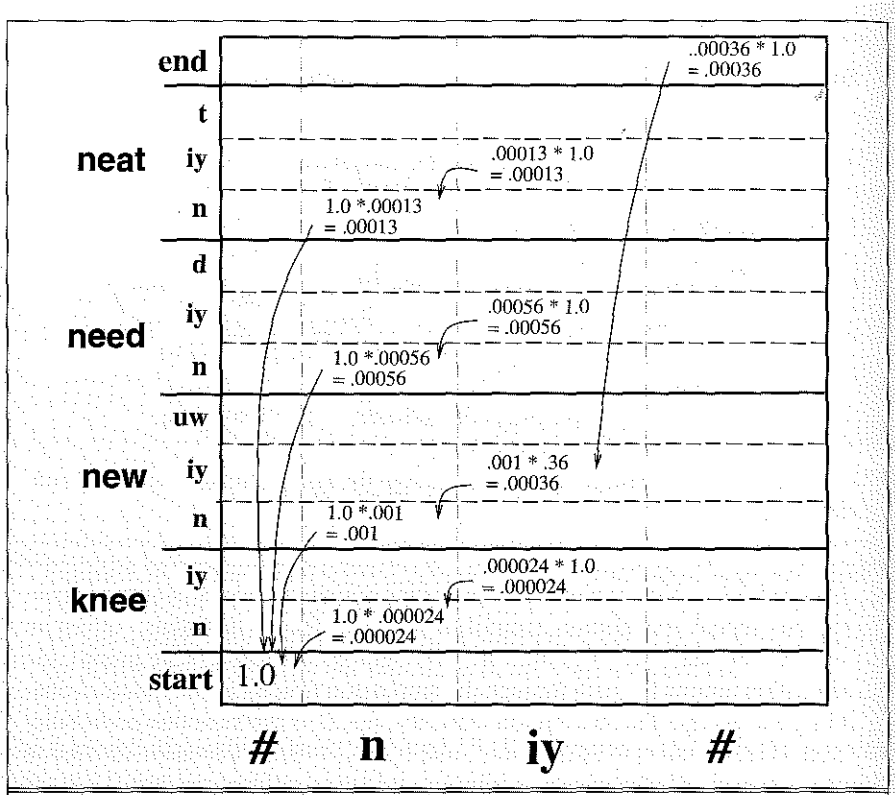


Figure 5.20 The entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. Backtracing from the *end* state, we can reconstruct the state sequence $n_{new}\ iy_{new}$, arriving at the best word *new*.

Weighted Automata and Segmentation

SEGMENTATION

Weighted automata and the Viterbi algorithm play an important in various algorithm for **segmentation**. Segmentation is the process of taking an undifferentiated sequence of symbols and "segmenting" it into chunks. For example **sentence segmentation** is the problem of automatically finding the sentence boundaries in a corpus. Similarly **word segmentation** is the problem of finding word-boundaries in a corpus. In written English there is no difficulty in segmenting words from each other because there are orthographic spaces between words. This is not the case in languages like Chinese and Japanese that use a Chinese-derived writing system. Written Chinese does not mark word boundaries. Instead, each Chinese character is written one after the other without spaces. Since each character approximately represents

a single morpheme, and since words can be composed of one or more characters, it is often difficult to know where words should be segmented. Proper word-segmentation is necessary for many applications, particularly including parsing and text-to-speech. (How a sentence is broken up into words influences its pronunciation in a number of ways.)

Consider the following example sentence from Sproat et al. (1996):

(5.15) 日文章鱼怎麽说？
"How do you say 'octopus' in Japanese?"

This sentence has two potential segmentations, only one of which is correct. In the plausible segmentation, the first two characters are combined to make the word for 'Japanese language' (日文 rì-wén) (the accents indicate the **tone** of each syllable), and the next two are combined to make the word for 'octopus' (章鱼 zhāng-yú).

(5.16)

日文	章鱼	怎麽	说	?
rì-wén	zhāng-yú	zěn-me	shuō	
Japanese	octopus	how	say	

"How do you say octopus in Japanese?"

(5.17)

日	文章	鱼	怎麽	说	?
rì	wén-zhāng	yú	zěn-me	shuō	
Japan	essay	fish	how	say	

"How do you say Japan essay fish?"

Sproat et al. (1996) give a very simple algorithm which selects the correct segmentation by choosing the one which contains the most-frequent words. In other words, the algorithm multiplies together the probabilities of each word in a potential segmentation and chooses whichever segmentation results in a higher product probability.

The implementation of their algorithm combines a weighted-finite-state transducer representation of a Chinese lexicon with the Viterbi algorithm. This lexicon is a slight augmentation of the FST lexicons we saw in Chapter 4; each word is represented as a series of arcs representing each character in the word, followed by a weighted arc representing the probability of the word. As is commonly true with probabilistic algorithms, they actually use the negative log probability of the word ($-\log(P(w))$). The log probability is mainly useful because the product of many probabilities gets very small, and so using the log probability can help avoid underflow. Using log probabilities also means that we are *adding costs* rather than *multiplying*

probabilities, and that we are looking for the *minimum cost* solution rather than the *maximum probability* solution.

Consider the example in Figure 5.21. This sample lexicon Figure 5.21(a) consists of only five potential words:

Word	Pronunciation	Meaning	Cost ($-logp(w)$)
日文	rì-wén	'Japanese'	10.63
日	rì	'Japan'	6.51
章鱼	zhāng- yú	'octopus'	13.18
文章	wén-zhāng	'essay'	9.51
鱼	yú	'fish'	10.28

The system represents the input sentence as the unweighted FSA in Figure 5.21(b). In order to compose this input with the lexicon, it needs to be converted into an FST. The algorithm uses a function *Id* which takes an FSA A and returns the FST which maps all and only the strings accepted by A to themselves. Let $D*$ represent the transitive closure of D, that is, the automaton created by adding a loop from the end of the lexicon back to the beginning. The set of all possible segmentations is $Id(I) \circ D^*$, that is, the input transducer $Id(I)$ composed with the transitive closure of the dictionary D, shown in Figure 5.21(c). Then the best segmentation is the lowest-cost segmentation in $Id(I) \circ D^*$, shown in Figure 5.21(d).

Finding the best path shown in Figure 5.21(d) can be done easily with the Viterbi algorithm and is left as an exercise for the reader. Furthermore, this segmentation algorithm, like the spelling error correction algorithm we saw earlier, can also be extended to incorporate the cross-word probabilities (N-gram probabilities) that will be introduced in Chapter 6.

Segmentation for Lexicon-Induction

The weighted automata segmentation algorithm that was presented above relies on the weights stored in the lexicon. But how is this lexicon to be learned in the first place? A number of segmentation algorithms address this "prior" problem of segmentation in the absence of a lexicon. For example de Marcken (1996) and Brent and Cartwright (1996) both propose algorithms that take an unsegmented sequence of input phones and use information-theoretic principles to iteratively induce the lexicon by trying different possible segmentations. Both rely on stochastic versions of the **Minimum Description Length (MDL)** principle and on phonotactic transition probabilities to choose between alternative models. The description length of a lexicon

MDL

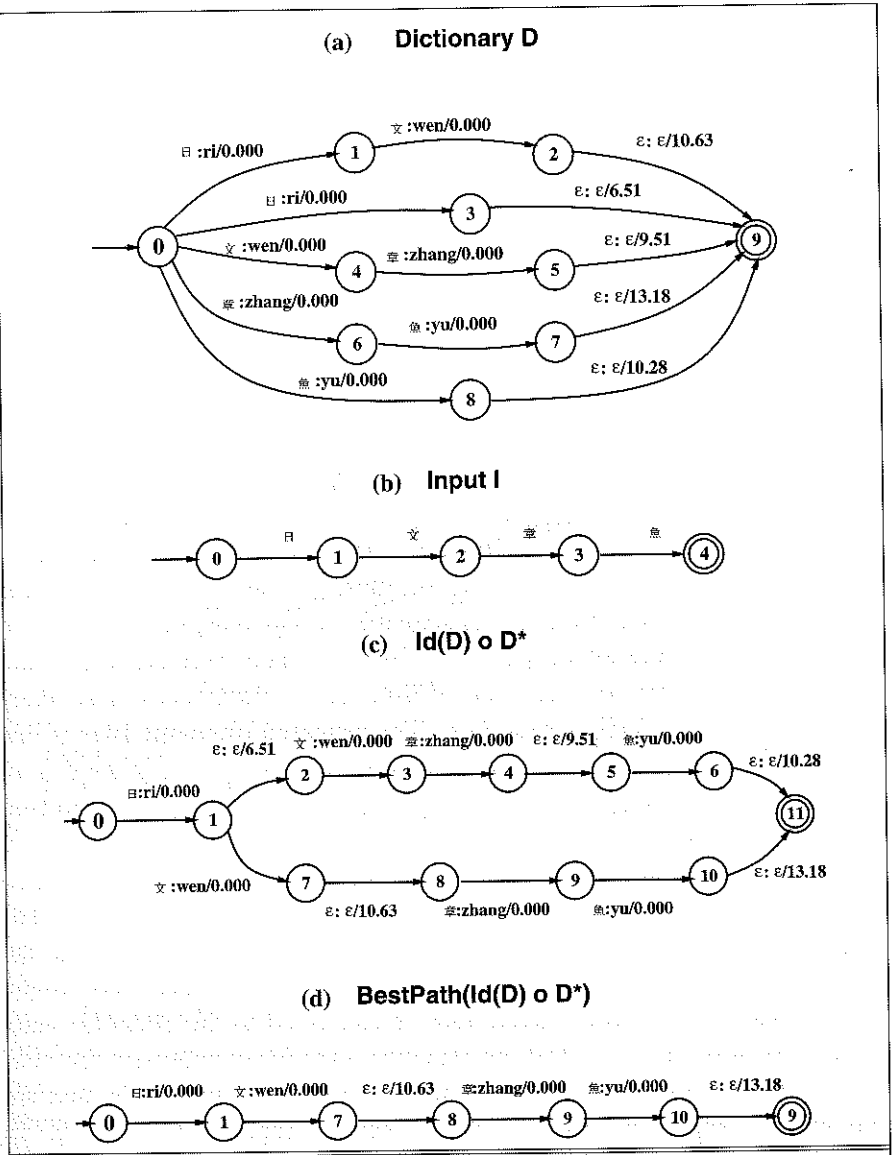


Figure 5.21 The Sproat et al. (1996) algorithm applied to four input words (after Sproat et al. (1996))

or grammar (measured, for example, in the number of symbols in it) is a heuristic measure of the information complexity in the lexicon. By preferring a lexicon with less symbols, MDL is implicitly choosing a simpler and

more general lexicon. Brent and Cartwright (1996) hypothesize that children use MDL algorithms to learn a lexicon by segmenting words from speech. In fact, Saffran et al. (1996) shows that eight-month-old infants can use phone sequence probabilities as evidence for word segmentation.

5.10 PRONUNCIATION IN HUMANS

Section 5.7 discussed many factors which influence pronunciation variation in humans. In this section we very briefly summarize a computational model of the retrieval of words from the mental lexicon as part of human lexical production. The model is due to Gary Dell and his colleagues; for brevity we combine and simplify features of multiple models (Dell, 1986, 1988; Dell et al., 1997) in this single overview. First consider some data. As we suggested in Chapter 3, production errors such as slips of the tongue (*darn bore* instead *barn door*) often provide important insights into lexical production. Dell (1986) summarizes a number of previous results about such slips. The **lexical bias** effect is that slips are more likely to create words than non-words; thus slips like *dean bad*→ *bean dad* are three times more likely than slips like *deal back*→ *beal dack*. The **repeated-phoneme bias** is that two phones in two words are likely to participate in an error if there is an identical phone in both words. Thus *deal beack* is more likely to slip to *beal* than *deal back* is.

The model that Dell (1986, 1988) proposes is a network with three levels: semantics, word (lemma), and phonemes.[6] The semantics level has nodes for concepts, the lemma level has one node for each words, and the phoneme level has separate nodes for each phone, separated into onsets, vowels, and codas. Each lemma node is connected to the phoneme units which comprise the word, and the semantic units which represent the concept. Connections are used to pass activation from node to node, and are bidirectional and excitatory. Lexical production happens in two stages. In the first stage, activation passes from the semantic concepts to words. Activation will cascade down into the phonological units and then back up into other word units. At some point the most highly activated word is selected. In the second stage, this selected is given a large jolt of activation. Again this activation passes to the phonological level. Now the most highly active phoneme nodes are selected and accessed in order.

[6] Dell (1988) also has a fourth level for syllable structure that we will ignore here.

Figure 5.22 shows Dell's model. Errors occur because too much activation reaches the wrong phonological node. Lexical bias, for example, is modeled by activation spreading up from the phones of the intended word to neighboring words, which then activated their own phones. Thus incorrect phones get "extra" activation if they are present in actual words.

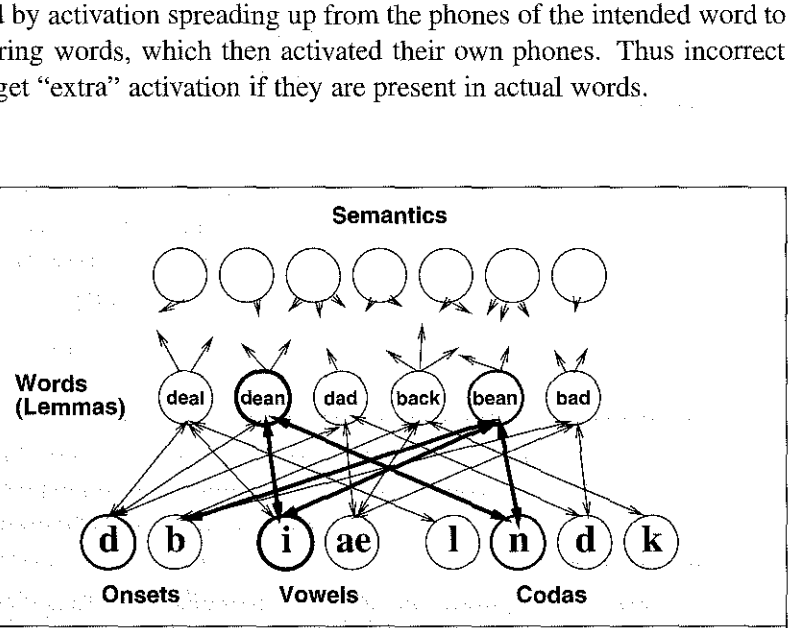


Figure 5.22 The network model of Dell (1986, 1988), showing the mechanism for lexical bias (modified from Dell (1988, p. 134)). The boldfaced nodes indicate nodes with lots of activation. The intended word *dean* has a greater chance of slipping to *bean* because of the existence of the *bean* node. The boldfaced lines show the connections which account for the possible slip.

The two-step network model also explains other facts about lexical production. **Aphasic** speakers have various troubles in language production and comprehension, often caused by strokes or accidents. Dell et al. (1997) show that weakening various connections in a network model like the one above can also account for the speech errors in aphasics. This supports the *continuity hypothesis*, which suggests that some part of aphasia is merely an extension of normal difficulties in word retrieval, and also provides further evidence for the network model. Readers interested in details of the model should see the above references and related computational models such as Roelofs (1997), which extends the network model to deal with syllabification, phonetic encoding, and more complex sequential structure, and Levelt et al. (1999).

APHASIC

5.11 SUMMARY

This chapter has introduced some essential metaphors and algorithms that will be useful throughout speech and language processing. The main points are as follows:

- We can represent many language problems as if a clean string of symbols had been corrupted by passing through a **noisy channel** and it is our job to recover the original symbol string. One powerful way to recover the original symbol string is to consider all possible original strings, and rank them by their **conditional probability**.
- The conditional probability is usually easiest to compute using the **Bayes Rule**, which breaks down the probability into a **prior** and a **likelihood**. For spelling error correction or pronunciation-modeling, the prior is computed by taking word frequencies or word bigram frequencies. The likelihood is computed by training a simple probabilistic model (like a confusion matrix, a decision tree, or a hand-written rule) on a database.
- The task of computing the distance between two strings comes up in spelling error correction and other problems. The **minimum edit distance** algorithm is an application of the **dynamic programming** paradigm to solving this problem, and can be used to produce the distance between two strings or an **alignment** of the two strings.
- The pronunciation of words is very variable. Pronunciation variation is caused by two classes of factors: **lexical variation** and **allophonic variation**. Lexical variation includes **sociolinguistic** factors like **dialect** and **register** or **style**.
- The single most important factor affecting allophonic variation is the identity of the surrounding phones. Other important factors include syllable structure, stress patterns, and the identity and frequency of the word.
- The **decoding** task is the problem of finding determining the correct "underlying" sequence of symbols that generated the "noisy" sequence of observation symbols.
- The **forward** algorithm is an efficient way of computing the likelihood of an observation sequence given a weighted automata. Like the **minimum edit distance** algorithm, it is a variant of dynamic programming. It will prove particularly in Chapter 7 when we consider Hidden

Markov Models, since it will allow us to sum multiple paths that each account for the same observation sequence.

- The **Viterbi** algorithm, another variant of dynamic programming, is an efficient way of solving the decoding problem by considering all possible strings and using the Bayes Rule to compute their probabilities of generating the observed "noisy" sequence.
- **Word segmentation** in languages without word-boundary markers, like Chinese and Japanese, is another kind of optimization task which can be solved by the Viterbi algorithm.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Algorithms for spelling error detection and correction have existing since at least Blair (1960). Most early algorithm were based on similarity keys like the Soundex algorithm discussed in the exercises on page 89 (Odell and Russell, 1922; Knuth, 1973). Damerau (1964) gave a dictionary-based algorithm for error detection; most error-detection algorithms since then have been based on dictionaries. Damerau also gave a correction algorithm that worked for single errors. Most algorithms since then have relied on dynamic programming, beginning with Wagner and Fischer (1974) (see below). Kukich (1992) is the definitive survey article on spelling error detection and correction. Only much later did probabilistic algorithms come into vogue for non-OCR spelling-error correction (for example Kashyap and Oommen (1983) and Kernighan et al. (1990)).

By contrast, the field of optical character recognition developed probabilistic algorithms quite early; Bledsoe and Browning (1959) developed a probabilistic approach to OCR spelling error correction that used a large dictionary and computed the likelihood of each observed letter sequence given each word in the dictionary by multiplying the likelihoods for each letter. In this sense Bledsoe and Browning also prefigured the modern Bayesian approaches to speech recognition. Shinghal and Toussaint (1979) and Hull and Srihari (1982) applied bigram letter-transition probabilities and the Viterbi algorithm to choose the most likely correct form for a misspelled OCR input.

The application of dynamic programming to the problem of sequence comparison has what Kruskal (1983) calls a "remarkable history of multiple

independent discovery and publication".[7] Kruskal and others give at least the following independently-discovered variants of the algorithm published in four separate fields:

Citation	Field
Viterbi (1967)	information theory
Vintsyuk (1968)	speech processing
Needleman and Wunsch (1970)	molecular biology
Sakoe and Chiba (1971)	speech processing
Sankoff (1972)	molecular biology
Reichert et al. (1973)	molecular biology
Wagner and Fischer (1974)	computer science

To the extent that there is any standard to terminology in speech and language processing, it is the use of the term **Viterbi** for the application of dynamic programming to any kind of probabilistic maximization problem. For non-probabilistic problems, the plain term **dynamic programming** is often used. The history of the forward algorithm, which derives from Hidden Markov Models, will be summarized in Chapter 7. Sankoff and Kruskal (1983) is a collection exploring the theory and use of sequence comparison in different fields. Forney (1973) is an early survey paper which explores the origin of the Viterbi algorithm in the context of information and communications theory.

The weighted finite-state automata was first described by Pereira et al. (1994), drawing from a combination of work in finite-state transducers and work in probabilistic languages (Booth and Thompson, 1973).

EXERCISES

5.1 Computing minimum edit distances by hand, figure out whether *drive* is closer to *brief* or to *divers*, and what the edit distance is. You may use any version of *distance* that you like.

5.2 Now implement a minimum edit distance algorithm and use your hand-computed results to check your code.

[7] Seven is pretty remarkable, but see page 15 for a discussion of the prevalence of multiple discovery.

5.3 The Viterbi algorithm can be used to extend a simplified version of the Kernighan et al. (1990) spelling error correction algorithm. Recall that the Kernighan et al. (1990) algorithm only allowed a single spelling error for each potential correction. Let's simplify by assuming that we only have three confusion matrices instead of four (*del*, *ins* and *sub*; no *trans*). Now show how the Viterbi algorithm can be used to extend the Kernighan et al. (1990) algorithm to handle multiple spelling errors per word.

5.4 To attune your ears to pronunciation reduction, listen for the pronunciation of the word *the*, *a*, or *to* in the spoken language around you. Try to notice when it is reduced, and mark down whatever facts about the speaker or speech situation that you can. What are your observations?

5.5 Find a speaker of a different dialect of English than your own (even someone from a slightly different region of your native dialect) and transcribe (using the ARPAbet or IPA) 10 words that they pronounce differently than you. Can you spot any generalizations?

5.6 Implement the Forward algorithm.

5.7 Write a modified version of the Viterbi algorithm which solves the segmentation problem from Sproat et al. (1996).

5.8 Now imagine a version of English that was written without spaces. Apply your segmentation program to this "compressed English". You will need other programs to compute word bigrams or trigrams.

5.9 Two words are **confusable** if they have phonetically similar pronunciations. Use one of your dynamic programming implementations to take two words and output a simple measure of how confusable they are. You will need to use an on-line pronunciation dictionary. You will also need a metric for how close together two phones are. Use your favorite set of phonetic feature vectors for this. You may assume some small constant probability of phone insertion and deletion. CONFUSABLE

6 N-GRAMS

But it must be recognized that the notion "probability of a sentence" is an entirely useless one, under any known interpretation of this term.

Noam Chomsky (1969, p. 57)

Anytime a linguist leaves the group the recognition rate goes up.
Fred Jelinek (then of the IBM speech group) (1988)[1]

Radar O'Reilly, the mild-mannered clerk of the 4077th M*A*S*H unit in the book, movie, and television show M*A*S*H, had an uncanny ability to guess what his interlocutor was about to say. Most of us don't have this skill, except perhaps when it comes to guessing the next words of songs written by very unimaginative lyricists. Or perhaps we do. For example what word is likely to follow this sentence fragment?

I'd like to make a collect...

Probably most of you concluded that a very likely word is *call*, although it's possible the next word could be *telephone*, or *person-to-person* or *international*. (Think of some others). The moral here is that guessing words is not as amazing as it seems, at least if we don't require perfect accuracy. Why is this important? Guessing the next word (or **word prediction**) is an essential subtask of speech recognition, hand-writing recognition, augmentative communication for the disabled, and spelling error detection. In

WORD PREDICTION

[1] In an address to the first Workshop on the Evaluation of Natural Language Processing Systems, December 7, 1988. While this workshop is described in Palmer and Finin (1990), the quote was not written down; some participants remember a more snappy version: *Every time I fire a linguist the performance of the recognizer improves.*

such tasks, word-identification is difficult because the input is very noisy and ambiguous. Thus looking at previous words can give us an important cue about what the next ones are going to be. Russell and Norvig (1995) give an example from *Take the Money and Run*, in which a bank teller interprets Woody Allen's sloppily written hold-up note as saying "I have a gub". A speech recognition system (and a person) can avoid this problem by their knowledge of word sequences ("a gub" isn't an English word sequence) and of their probabilities (especially in the context of a hold-up, "I have a gun" will have a much higher probability than "I have a gub" or even "I have a gull").

AUGMENTATIVE COMMUNICATION

This ability to predict the next word is important for **augmentative communication** systems (Newell et al., 1998). These are computer systems that help the disabled in communication. For example, people who are unable to use speech or sign-language to communicate, like the physicist Steven Hawking, use systems that speak for them, letting them choose words with simple hand movements, either by spelling them out, or by selecting from a menu of possible words. But spelling is very slow, and a menu of words obviously can't have all possible English words on one screen. Thus it is important to be able to know which words the speaker is likely to want to use next, so as to put those on the menu.

Finally, consider the problem of detecting real-word spelling errors. These are spelling errors that result in real English words (although not the ones the writer intended) and so detecting them is difficult (we can't find them by just looking for words that aren't in the dictionary). Figure 6.1 gives some examples.

They are leaving in about fifteen *minuets* to go to her house.
The study was conducted mainly *be* John Black.
The design *an* construction of the system will take more than a year.
Hopefully, all *with* continue smoothly in my absence.
Can they *lave* him my messages?
I need to *notified* the bank of [this problem.]
He is trying to *fine* out.

Figure 6.1 Some attested real-word spelling errors from Kukich (1992).

These errors can be detected by algorithms which examine, among other features, the words surrounding the errors. For example, while the phrase *in about fifteen minuets* is perfectly grammatical English, it is a very

unlikely combination of words. Spellcheckers can look for low probability combinations like this. In the examples above the probability of three word combinations (*they lave him*, *to fine out*, *to notified the*) is very low. Of course sentences with no spelling errors may also have low probability word sequences, which makes the task challenging. We will see in Section 6.6 that there are a number of different machine learning algorithms which make use of the surrounding words and other features to do **context-sensitive spelling error correction**.

Guessing the next word turns out to be closely related to another problem: computing the probability of a sequence of words. For example the following sequence of words has a non-zero probability of being encountered in a text written in English:

> ... all of a sudden I notice three guys standing on the sidewalk taking a very good long gander at me.

while this same set of words in a different order probably has a very low probability:

> good all I of notice a taking sidewalk the me long three at sudden guys gander on standing a a the very

Algorithms that assign a probability to a sentence can also be used to assign a probability to the next word in an incomplete sentence, and vice versa. We will see in later chapters that knowing the probability of whole sentences or strings of words is useful in part-of-speech-tagging (Chapter 8), word-sense disambiguation, and probabilistic parsing Chapter 12.

This model of word prediction that we will introduce in this chapter is the **N-gram**. An N-gram model uses the previous $N-1$ words to predict the next one. In speech recognition, it is traditional to use the term **language model** or **LM** for such statistical models of word sequences. In the rest of this chapter we will be using both **language model** and **grammar**, depending on the context.

N-GRAM

LANGUAGE MODEL

LM

6.1 Counting Words in Corpora

> [upon being asked if there weren't enough words in the English language for him]:
> *"Yes, there are enough, but they aren't the right ones."*
> James Joyce, reported in Bates (1997)

Probabilities are based on counting things. Before we talk about probabilities, we need to decide what we are going to count and where we are going to find the things to count.

CORPORA
CORPUS

As we saw in Chapter 5, statistical processing of natural language is based on **corpora** (singular **corpus**), on-line collections of text and speech. For computing word probabilities, we will be counting words in a training corpus. Let's look at part of the Brown Corpus, a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.), which was assembled at Brown University in 1963-64 (Kučera and Francis, 1967; Francis, 1979; Francis and Kučera, 1982). It contains sentence (6.1); how many words are in this sentence?

(6.1) He stepped out into the hall, was delighted to encounter a water brother.

Example (6.1) has 13 words if we don't count punctuation-marks as words, 15 if we count punctuation. Whether we treat period (".), comma (","), and so on as words depends on the task. There are tasks such as grammar-checking, spelling error detection, or author-identification, for which the location of the punctuation is important (for checking for proper capitalization at the beginning of sentences, or looking for interesting patterns of punctuation usage that uniquely identify an author). In natural language processing applications, question-marks are an important cue that someone has asked a question. Punctuation is a useful cue for part-of-speech tagging. These applications, then, often count punctuation as words.

Unlike text corpora, corpora of spoken language usually don't have punctuation, but speech corpora do have other phenomena that we might or might not want to treat as words. One speech corpus, the Switchboard corpus of telephone conversations between strangers, was collected in the early 1990s and contains 2430 conversations averaging 6 minutes each, for a total of 240 hours of speech and 3 million words (Godfrey et al., 1992). Here's a sample utterance of Switchboard (since the units of spoken language are different than written language, we will use the word **utterance** rather than "sentence" when we are referring to spoken language):

UTTERANCE

(6.2) I do uh main- mainly business data processing

FRAGMENTS
FILLED PAUSES

This utterance, like many or most utterances in spoken language, has **fragments**, words that are broken off in the middle, like the first instance of the word *mainly*, represented here as *main-*. It also has **filled pauses** like *uh*, which don't occur in written English. Should we consider these to be words? Again, it depends on the application. If we are building an automatic

dictation system based on automatic speech recognition, we might want to strip out the fragments. But the *uh*s and *um*s are in fact much more like words. For example, Smith and Clark (1993) and Clark (1994) have shown that *um* has a slightly different meaning than *uh* (generally speaking *um* is used when speakers are having major planning problems in producing an utterance, while *uh* is used when they know what they want to say, but are searching for the exact words to express it). Stolcke and Shriberg (1996b) also found that *uh* can be a useful cue in predicting the next word (why might this be?), and so most speech recognition systems treat *uh* as a word.

Are capitalized tokens like *They* and uncapitalized tokens like *they* the same word? For most statistical applications these are lumped together, although sometimes (for example for spelling error correction or part-of-speech-tagging) the capitalization is retained as a separate feature. For the rest of this chapter we will assume our models are not case-sensitive.

How should we deal with inflected forms like *cats* versus *cat*? Again, this depends on the application. Most current N-gram based systems are based on the **wordform**, which is the inflected form as it appears in the corpus. Thus these are treated as two separate words. This is not a good simplification for many domains, which might want to treat *cats* and *cat* as instances of a single abstract word, or **lemma**. A lemma is a set of lexical forms having the same stem, the same major part-of-speech, and the same word-sense. We will return to the distinction between wordforms (which distinguish *cat* and *cats*) and lemmas (which lump *cat* and *cats* together) in Chapter 16.

WORDFORM

LEMMA

How many words are there in English? One way to answer this question is to count in a corpus. We use **types** to mean the number of distinct words in a corpus, that is, the size of the vocabulary, and **tokens** to mean the total number of running words. Thus the following sentence from the Brown corpus has 16 word tokens and 14 word types (not counting punctuation):

TYPES

TOKENS

(6.3) They picnicked by the pool, then lay back on the grass and looked at the stars.

The Switchboard corpus has 2.4 million wordform tokens and approximately 20,000 wordform types. This includes proper nouns. Spoken language is less rich in its vocabulary than written language: Kučera (1992) gives a count for Shakespeare's complete works at 884,647 wordform tokens from 29,066 wordform types. Thus each of the 884,647 wordform tokens is a repetition of one of the 29,066 wordform types. The 1 million wordform tokens of the Brown corpus contain 61,805 wordform types that belong to

37,851 lemma types. All these corpora are quite small. Brown et al. (1992) amassed a corpus of 583 million wordform tokens of English that included 293,181 different wordform types.

Dictionaries are another way to get an estimate of the number of words, although since dictionaries generally do not include inflected forms they are better at measuring lemmas than wordforms. The American Heritage third edition dictionary has 200,000 "boldface forms"; this is somewhat higher than the true number of lemmas, since there can be one or more boldface form per lemma (and since the boldface forms includes multiword phrases).

The rest of this chapter will continue to distinguish between types and tokens. "Types" will mean wordform types and not lemma types, and punctuation marks will generally be counted as words.

6.2 SIMPLE (UNSMOOTHED) *N*-GRAMS

The models of word sequences we will consider in this chapter are probabilistic models; ways to assign probabilities to strings of words, whether for computing the probability of an entire sentence or for giving a probabilistic prediction of what the next word will be in a sequence. As we did in Chapter 5, we will assume that the reader has a basic knowledge of probability theory.

The simplest possible model of word sequences would simply let any word of the language follow any other word. In the probabilistic version of this theory, then, every word would have an equal probability of following every other word. If English had 100,000 words, the probability of any word following any other word would be $\frac{1}{100,000}$ or .00001.

In a slightly more complex model of word sequences, any word could follow any other word, but the following word would appear with its normal frequency of occurrence. For example, the word *the* has a high relative frequency, it occurs 69,971 times in the Brown corpus of 1,000,000 words (i.e., 7% of the words in this particular corpus are *the*). By contrast the word *rabbit* occurs only 11 times in the Brown corpus.

We can use these relative frequencies to assign a probability distribution across following words. So if we've just seen the string *Anyhow,* we can use the probability .07 for *the* and .00001 for *rabbit* to guess the next word. But suppose we've just seen the following string:

Just then, the white

In this context *rabbit* seems like a more reasonable word to follow *white* than *the* does. This suggests that instead of just looking at the individual relative frequencies of words, we should look at the conditional probability of a word given the previous words. That is, the probability of seeing *rabbit* given that we just saw *white* (which we will represent as $P(rabbit|white)$) is higher than the probability of *rabbit* otherwise.

Given this intuition, let's look at how to compute the probability of a complete string of words (which we can represent either as $w_1 \ldots w_n$ or w_1^n). If we consider each word occurring in its correct location as an independent event, we might represent this probability as follows:

$$P(w_1, w_2 \ldots, w_{n-1}, w_n) \tag{6.4}$$

We can use the chain rule of probability to decompose this probability:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)\ldots P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned} \tag{6.5}$$

But how can we compute probabilities like $P(w_n|w_1^{n-1})$? We don't know any easy way to compute the probability of a word given a long sequence of preceding words. (For example, we can't just count the number of times every word occurs following every long string; we would need far too large a corpus).

We solve this problem by making a useful simplification: we *approximate* the probability of a word given all the previous words. The approximation we will use is very simple: the probability of the word given the single previous word! The **bigram** model approximates the probability of a word given all the previous words $P(w_n|w_1^{n-1})$ by the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

BIGRAM

$$P(\text{rabbit}|\text{Just the other I day I saw a}) \tag{6.6}$$

we approximate it with the probability

$$P(\text{rabbit}|\text{a}) \tag{6.7}$$

This assumption that the probability of a word depends only on the previous word is called a **Markov** assumption. Markov models are the class of probabilistic models that assume that we can predict the probability of some future unit without looking too far into the past. We saw this use of the word **Markov** in introducing the **Markov chain** in Chapter 5. Recall that a

MARKOV

Markov chain is a kind of weighted finite-state automaton; the intuition of the term Markov in Markov chain is that the next state of a weighted FSA is always dependent on a finite history (since the number of states in a finite-state automaton is finite). The basic bigram model can be viewed as a simple kind of Markov chain which has one state for each word.

N-GRAM
FIRST-ORDER
SECOND-ORDER

We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the **N-gram** (which looks $N-1$ words into the past). A bigram is called a **first-order** Markov model (because it looks one token into the past), a trigram is a **second-order** Markov model, and in general an N-gram is a $N-1$th order Markov model. Markov models of words were common in engineering, psychology, and linguistics until Chomsky's influential review of Skinner's *Verbal Behavior* in 1958 (see the History section at the back of the chapter), but went out of vogue until the success of N-gram models in the IBM speech recognition laboratory at the Thomas J. Watson Research Center. brought them back to the attention of the community.

The general equation for this N-gram approximation to the conditional probability of the next word in a sequence is:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}) \tag{6.8}$$

Equation 6.8 shows that the probability of a word w_n given all the previous words can be approximated by the probability given only the previous N words.

For a bigram grammar, then, we compute the probability of a complete string by substituting Equation (6.8) into Equation (6.5). The result:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{6.9}$$

Let's look at an example from a speech-understanding system. The Berkeley Restaurant Project is a speech-based restaurant consultant; users ask questions about restaurants in Berkeley, California, and the system displays appropriate information from a database of local restaurants (Jurafsky et al., 1994). Here are some sample user queries:

I'm looking for Cantonese food.
I'd like to eat dinner someplace nearby.
Tell me about Chez Panisse.
Can you give me a listing of the kinds of food that are available?
I'm looking for a good place to eat breakfast.
I definitely do not want to have cheap Chinese food.

When is Caffe Venezia open during the day?
I don't wanna walk more than ten minutes.

Table 6.2 shows a sample of the bigram probabilities for some of the words that can follow the word *eat*, taken from actual sentences spoken by users (putting off just for now the algorithm for training bigram probabilities). Note that these probabilities encode some facts that we think of as strictly syntactic in nature (like the fact that what comes after *eat* is usually something that begins a noun phrase, that is, an adjective, quantifier or noun), as well as facts that we think of as more culturally based (like the low probability of anyone asking for advice on finding British food).

eat on	.16	eat Thai	.03
eat some	.06	eat breakfast	.03
eat lunch	.06	eat in	.02
eat dinner	.05	eat Chinese	.02
eat at	.04	eat Mexican	.02
eat a	.04	eat tomorrow	.01
eat Indian	.04	eat dessert	.007
eat today	.03	eat British	.001

Figure 6.2 A fragment of a bigram grammar from the Berkeley Restaurant Project showing the most likely words to follow *eat*.

Assume that in addition to the probabilities in Table 6.2, our grammar also includes the bigram probabilities in Table 6.3 (<s> is a special word meaning "Start of sentence").

<s> I	.25	I want	.32	want to	.65	to eat	.26	British food	.60
<s> I'd	.06	I would	.29	want a	.05	to have	.14	British restaurant	.15
<s> Tell	.04	I don't	.08	want some	.04	to spend	.09	British cuisine	.01
<s> I'm	.02	I have	.04	want thai	.01	to be	.02	British lunch	.01

Figure 6.3 More fragments from the bigram grammar from the Berkeley Restaurant Project.

Now we can compute the probability of sentences like *I want to eat British food* or *I want to eat Chinese food* by simply multiplying the appropriate bigram probabilities together, as follows:

$$\begin{aligned} P(\text{I want to eat British food}) &= P(\text{I}|\text{<s>})P(\text{want}|\text{I})P(\text{to}|\text{want}) \\ &\quad P(\text{eat}|\text{to})P(\text{British}|\text{eat}) \\ &\quad P(\text{food}|\text{British}) \end{aligned}$$

$$= .25 * .32 * .65 * .26 * .002 * .60$$
$$= .000016$$

As we can see, since probabilities are all less than 1 (by definition), the product of many probabilities gets smaller the more probabilities we multiply. This causes a practical problem: the risk of numerical underflow. If we are computing the probability of a very long string (like a paragraph or an entire document) it is more customary to do the computation in log space; we take the log of each probability (the **logprob**), add all the logs (since adding in log space is equivalent to multiplying in linear space) and then take the anti-log of the result. For this reason many standard programs for computing N-grams actually store and calculate all probabilities as logprobs. In this text we will always report logs in base 2 (i.e., we will use log to mean $\log_2$).

LOGPROB

TRIGRAM

A **trigram** model looks just the same as a bigram model, except that we condition on the two previous words (e.g., we use $P(food|eat\ British)$ instead of $P(\text{food}|\text{British})$). To compute trigram probabilities at the very beginning of sentence, we can use two pseudo-words for the first trigram (i.e., $P(I|<start1><start2>)$).

NORMALIZING

N-gram models can be trained by counting and **normalizing** (for probabilistic models, normalizing means dividing by some total count so that the resulting probabilities fall legally between 0 and 1). We take some training corpus, and from this corpus take the count of a particular bigram, and divide this count by the sum of all the bigrams that share the same first word:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{6.10}$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1}. (The reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{6.11}$$

For the general case of N-gram parameter estimation:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})} \tag{6.12}$$

Equation 6.12 estimates the N-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a **relative frequency**; the use of relative frequencies as a way to estimate probabilities is one example of the technique known as **Maximum Likelihood Estimation** or **MLE**, because the resulting

RELATIVE FREQUENCY

MAXIMUM LIKELIHOOD ESTIMATION

MLE

parameter set is one in which the likelihood of the training set T given the model M (i.e., $P(T|M)$) is maximized. For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that it will occur in some other text of way a million words? The MLE estimate of its probability is $\frac{400}{1000000}$ or .0004. Now .0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; but it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus.

There are better methods of estimating N-gram probabilities than using relative frequencies (we will consider a class of important algorithms in Section 6.3), but even the more sophisticated algorithms make use in some way of this idea of relative frequency. Figure 6.4 shows the bigram counts from a piece of a bigram grammar from the Berkeley Restaurant Project. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of seven words would be even more sparse.

	I	want	to	eat	Chinese	food	lunch
I	8	1087	0	13	0	0	0
want	3	0	786	0	6	8	6
to	3	0	10	860	3	0	12
eat	0	0	2	0	19	2	52
Chinese	2	0	0	0	0	120	1
food	19	0	17	0	0	0	0
lunch	4	0	0	0	0	1	0

Figure 6.4 Bigram counts for seven of the words (out of 1616 total word types) in the Berkeley Restaurant Project corpus of ≈10,000 sentences.

Figure 6.5 shows the bigram probabilities after normalization (dividing each row by the following appropriate unigram counts):

I	3437
want	1215
to	3256
eat	938
Chinese	213
food	1506
lunch	459

	I	want	to	eat	Chinese	food	lunch
I	.0023	.32	0	.0038	0	0	0
want	.0025	0	.65	0	.0049	.0066	.0049
to	.00092	0	.0031	.26	.00092	0	.0037
eat	0	0	.0021	0	.020	.0021	.055
Chinese	.0094	0	0	0	0	.56	.0047
food	.013	0	.011	0	0	0	0
lunch	.0087	0	0	0	0	.0022	0

Figure 6.5 Bigram probabilities for seven of the words (out of 1616 total word types) in the Berkeley Restaurant Project corpus of ≈10,000 sentences.

More on N-grams and Their Sensitivity to the Training Corpus

In this section we look at a few examples of different N-gram models to get an intuition for two important facts about their behavior. The first is the increasing accuracy of N-gram models as we increase the value of N. The second is their very strong dependency on their training corpus (in particular its genre and its size in words).

We do this by borrowing a visualization technique proposed by Shannon (1951) and also used by Miller and Selfridge (1950). The idea is to train various N-grams and then use each to generate random sentences. It's simplest to visualize how this works for the unigram case. Imagine all the words of English covering the probability space between 0 and 1. We choose a random number between 0 and 1, and print out the word that covers the real value we have chosen. The same technique can be used to generate higher order N-grams by first generating a random bigram that starts with <s> (according to its bigram probability), then choosing a random bigram to follow it (again, where the likelihood of following a particular bigram is proportional to its conditional probability), and so on.

To give an intuition for the increasing power of higher order N-grams, we trained a unigram, bigram, trigram, and a quadrigram model on the complete corpus of Shakespeare's works. We then used these four grammars to generate random sentences. In the following examples we treated each punctuation mark as if it were a word in its own right, and we trained the grammars on a version of the corpus with all capital letters changed to lowercase. After generated the sentences we corrected the output for capitalization just to improve readability. Some of the resulting sentences:

1. Unigram approximation to Shakespeare

(a) To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
(b) Every enter now severally so, let
(c) Hill he late speaks; or! a more to leg less first you enter
(d) Will rash been and by I the me loves gentle me not slavish page, the and hour; ill let
(e) Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

2. Bigram approximation to Shakespeare

(a) What means, sir. I confess she? then all sorts, he is trim, captain.
(b) Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
(c) What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?
(d) Enter Menenius, if it so many good direction found'st thou art a strong upon command of fear not a liberal largess given away, Falstaff! Exeunt
(e) Thou whoreson chops. Consumption catch your dearest friend, well, and I know where many mouths upon my undoing all but be, how soon, then; we'll execute upon my love's bonds and we do you will?
(f) The world shall- my lord!

3. Trigram approximation to Shakespeare

(a) Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
(b) This shall forbid it should be branded, if renown made it empty.
(c) What is't that cried?
(d) Indeed the duke; and had a very good friend.
(e) Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.
(f) The sweet! How many then shall posthumus end his miseries.

4. Quadrigram approximation to Shakespeare

(a) King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
(b) Will you not tell me who I am?
(c) It cannot be but so.
(d) Indeed the short and the long. Marry, 'tis a noble Lepidus.
(e) They say all lovers swear more performance than they are wont to keep obliged faith unforfeited!
(f) Enter Leonato's brother Antonio, and the rest, but seek the weary beds of people sick.

METHODOLOGY BOX: TRAINING SETS AND TEST SETS

The probabilities in a statistical model like an N-gram come from the corpus it is trained on. This **training corpus** needs to be carefully designed. If the training corpus is too specific to the task or domain, the probabilities may be too narrow and not generalize well to new sentences. If the training corpus is too general, the probabilities may not do a sufficient job of reflecting the task or domain.

Furthermore, suppose we are trying to compute the probability of a particular "test" sentence. If our "'test' sentence is part of the training corpus, it will have an artificially high probability. The training corpus must not be biased by including this sentence. Thus when using a statistical model of language given some corpus of relevant data, we start by dividing the data into a **training set** and a **test set**. We train the statistical parameters of the model on the training set, and then use them to compute probabilities on the test set.

This training-and-testing paradigm can also be used to **evaluate** different N-gram architectures. For example to compare the different **smoothing** algorithms we will introduce in Section 6.3, we can take a large corpus and divide it into a training set and a test set. Then we train the two different N-gram models on the training set and see which one better models the test set. But what does it mean to "model the test set"? There is a useful metric for how well a given statistical model matches a test corpus, called **perplexity**. Perplexity is a variant of **entropy**, and will be introduced on page 223.

In some cases we need more than one test set. For example, suppose we have a few different possible language models and we want first to pick the best one and then to see how it does on a fair test set, that is, one we've never looked at before. We first use a **development test set** (also called a **devtest** set) to pick the best language model, and perhaps tune some parameters. Then once we come up with what we think is the best model, we run it on the true test set.

When comparing models it is important to use statistical tests (introduced in any statistics class or textbook for the social sciences) to determine if the difference between two models is significant. Cohen (1995) is a useful reference which focuses on statistical research methods for artificial intelligence. Dietterich (1998) focuses on statistical tests for comparing classifiers.

The longer the context on which we train the model, the more coherent the sentences. In the unigram sentences, there is no coherent relation between words, and in fact none of the sentences end in a period or other sentence-final punctuation. The bigram sentences can be seen to have very local word-to-word coherence (especially if we consider that punctuation counts as a word). The trigram and quadrigram sentences are beginning to look a lot like Shakespeare. Indeed a careful investigation of the quadrigram sentences shows that they look a little too much like Shakespeare. The words *It cannot be but so* are directly from *King John*. This is because the Shakespeare oeuvre, while large by many standards, is somewhat less than a million words. Recall that Kučera (1992) gives a count for Shakespeare's complete works at 884,647 words (tokens) from 29,066 wordform types (including proper nouns). That means that even the bigram model is very sparse; with $29,066$ types, there are $29,066^2$, or more than 844 million possible bigrams, so a 1 million word training set is clearly vastly insufficient to estimate the frequency of the rarer ones; indeed somewhat under 300,000 different bigram types actually occur in Shakespeare. This is far too small to train quadrigrams; thus once the generator has chosen the first quadrigram (*It cannot be but*), there are only five possible continuations (*that*, *I*, *he*, *thou*, and *so*); indeed for many quadrigrams there is only one continuation.

To get an idea of the dependence of a grammar on its training set, let's look at an N-gram grammar trained on a completely different corpus: the Wall Street Journal (WSJ). A native speaker of English is capable of reading both Shakespeare and the Wall Street Journal; both are subsets of English. Thus it seems intuitive that our N-grams for Shakespeare should have some overlap with N-grams from the Wall Street Journal. In order to check whether this is true, here are three sentences generated by unigram, bigram, and trigram grammars trained on 40 million words of articles from the daily Wall Street Journal (these grammars are Katz backoff grammars with Good-Turing smoothing; we will learn in the next section how these are constructed). Again, we have corrected the output by hand with the proper English capitalization for readability.

1. (*unigram*) Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2. (*bigram*) Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep

her

3. (*trigram*) They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Compare these examples to the pseudo-Shakespeare on the previous page; while superficially they both seem to model "English-like sentences" there is obviously no overlap whatsoever in possible sentences, and very little if any overlap even in small phrases. The difference between the Shakespeare and WSJ corpora tell us that a good statistical approximation to English will have to involve a very large corpus with a very large cross-section of different genres. Even then a simple statistical model like an N-gram would be incapable of modeling the consistency of style across genres. (We would only want to expect Shakespearean sentences when we are reading Shakespeare, not in the middle of a Wall Street Journal article.)

6.3 SMOOTHING

Never do I ever want
to hear another word!
There isn't one,
I haven't heard!
Eliza Doolittle in Alan Jay Lerner's *My Fair Lady* lyrics

words people
never use —
could be
only I
know them
Ishikawa Takuboku 1885–1912

One major problem with standard N-gram models is that they must be trained from some corpus, and because any particular training corpus is finite, some perfectly acceptable English N-grams are bound to be missing from it. That is, the bigram matrix for any given training corpus is **sparse**; it is bound to have a very large number of cases of putative "zero probability bigrams" that should really have some non-zero probability. Furthermore,

SPARSE

the MLE method also produces poor estimates when the counts are non-zero but still small.

Some part of this problem is endemic to N-grams; since they can't use long-distance context, they always tend to underestimate the probability of strings that happen not to have occurred nearby in their training corpus. But there are some techniques we can use to assign a non-zero probability to these "zero probability bigrams". This task of reevaluating some of the zero-probability and low-probability N-grams, and assigning them non-zero values, is called **smoothing**. In the next few sections we will introduce some smoothing algorithms and show how they modify the Berkeley Restaurant bigram probabilities in Figure 6.5.

SMOOTHING

Add-One Smoothing

One simple way to do smoothing might be just to take our matrix of bigram counts, before we normalize them into probabilities, and add one to all the counts. This algorithm is called **add-one** smoothing. Although this algorithm does not perform well and is not commonly used, it introduces many of the concepts that we will see in other smoothing algorithms, and also gives us a useful baseline.

ADD-ONE

Let's first consider the application of add-one smoothing to unigram probabilities, since that will be simpler. The unsmoothed maximum likelihood estimate of the unigram probability can be computed by dividing the count of the word by the total number of word tokens N:

$$P(w_x) = \frac{c(w_x)}{\sum_i c(w_i)} = \frac{c(w_x)}{N}$$

The various smoothing estimates will rely on an adjusted count c^*. The count adjustment for add-one smoothing can then be defined by adding one to the count and then multiplying by a normalization factor, $\frac{N}{N+V}$, where V is the total number of word types in the language, that is, the **vocabulary size**. Since we are adding 1 to the count for each word type, the total number of tokens must be increased by the number of types. The adjusted count for add-one smoothing is then defined as:

VOCABULARY SIZE

$$c_i^* = (c_i + 1)\frac{N}{N+V} \tag{6.13}$$

and the counts can be turned into probabilities p_i^* by normalizing by N.

DISCOUNTING

An alternative way to view a smoothing algorithm is as **discounting** (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts. Thus instead of referring to the discounted counts c^*, many papers also define smoothing algorithms in terms of a **discount** d_c, the ratio of the discounted counts to the original counts:

DISCOUNT

$$d_c = \frac{c^*}{c}$$

Alternatively, we can compute the probability p_i^* directly from the counts as follows:

$$p_i^* = \frac{c_i + 1}{N + V}$$

Now that we have the intuition for the unigram case, let's smooth our Berkeley Restaurant Project bigram. Figure 6.6 shows the add-one-smoothed counts for the bigram in Figure 6.4.

	I	want	to	eat	Chinese	food	lunch
I	9	1088	1	14	1	1	1
want	4	1	787	1	7	9	7
to	4	1	11	861	4	1	13
eat	1	1	3	1	20	3	53
Chinese	3	1	1	1	1	121	2
food	20	1	18	1	1	1	1
lunch	5	1	1	1	1	2	1

Figure 6.6 Add-one Smoothed Bigram counts for seven of the words (out of 1616 total word types) in the Berkeley Restaurant Project corpus of ≈10,000 sentences.

Figure 6.7 shows the add-one-smoothed probabilities for the bigram in Figure 6.5. Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{6.14}$$

For add-one-smoothed bigram counts we need to first augment the unigram count by the number of total word types in the vocabulary V:

$$p^*(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V} \tag{6.15}$$

We need to add V (= 1616) to each of the unigram counts:

I	3437+1616	= 5053
want	1215+1616	= 2931
to	3256+1616	= 4872
eat	938+1616	= 2554
Chinese	213+1616	= 1829
food	1506+1616	= 3122
lunch	459+1616	= 2075

The result is the smoothed bigram probabilities in Figure 6.7.

	I	want	to	eat	Chinese	food	lunch
I	.0018	.22	.00020	.0028	.00020	.00020	.00020
want	.0014	.00035	.28	.00035	.0025	.0032	.0025
to	.00082	.00021	.0023	.18	.00082	.00021	.0027
eat	.00039	.00039	.0012	.00039	.0078	.0012	.021
Chinese	.0016	.00055	.00055	.00055	.00055	.066	.0011
food	.0064	.00032	.0058	.00032	.00032	.00032	.00032
lunch	.0024	.00048	.00048	.00048	.00048	.00096	.00048

Figure 6.7 Add-one smoothed bigram probabilities for seven of the words (out of 1616 total word types) in the Berkeley Restaurant Project corpus of ≈10,000 sentences.

It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts. These adjusted counts can be computed by Equation (6.13). Figure 6.8 shows the reconstructed counts.

Note that add-one smoothing has made a very big change to the counts. $C(\textit{want to})$ changed from 786 to 331! We can see this in probability space as well: $P(\textit{to}|\textit{want})$ decreases from .65 in the unsmoothed case to .28 in the smoothed case.

Looking at the discount d (the ratio between new and old counts) shows us how strikingly the counts for each prefix-word have been reduced; the bigrams starting with *Chinese* were discounted by a factor of 8!

	I	want	to	eat	Chinese	food	lunch
I	6	740	.68	10	.68	.68	.68
want	2	.42	331	.42	3	4	3
to	3	.69	8	594	3	.69	9
eat	.37	.37	1	.37	7.4	1	20
Chinese	.36	.12	.12	.12	.12	15	.24
food	10	.48	9	.48	.48	.48	.48
lunch	1.1	.22	.22	.22	.22	.44	.22

Figure 6.8 Add-one smoothed bigram counts for seven of the words (out of 1616 total word types) in the Berkeley Restaurant Project Corpus of ≈10,000 sentences.

I	.68
want	.42
to	.69
eat	.37
Chinese	.12
food	.48
lunch	.22

The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros. The problem is that we arbitrarily picked the value "1" to add to each count. We could avoid this problem by adding smaller values to the counts ("add-one-half" "add-one-thousandth"), but we would need to retrain this parameter for each situation.

In general add-one smoothing is a poor method of smoothing. Gale and Church (1994) summarize a number of additional problems with the add-one method; the main problem is that add-one is much worse at predicting the actual probability for bigrams with zero counts than other methods like the Good-Turing method we will describe below. Furthermore, they show that variances of the counts produced by the add-one method are actually worse than those from the unsmoothed MLE method.

Witten-Bell Discounting

WITTEN-BELL DISCOUNTING

A much better smoothing algorithm that is only slightly more complex than Add-One smoothing we will refer to as **Witten-Bell discounting** (it is introduced as Method C in Witten and Bell (1991)). Witten-Bell discounting is based on a simple but clever intuition about zero-frequency events. Let's think of a zero-frequency word or N-gram as one that just hasn't happened

yet. When it does happen, it will be the first time we see this new N-gram. So the probability of seeing a zero-frequency N-gram can be modeled by the probability of seeing an N-gram for the first time. This is a recurring concept in statistical language processing:

Key Concept #4. Things Seen Once: Use the count of things you've seen once to help estimate the count of things you've never seen.

The idea that we can estimate the probability of "things we never saw" with help from the count of "things we saw once" will return when we discuss Good-Turing smoothing later in this chapter, and then once again when we discuss methods for tagging an unknown word with a part-of-speech in Chapter 8.

How can we compute the probability of seeing an N-gram for the first time? By counting the number of times we saw N-grams for the first time in our training corpus. This is very simple to produce since the count of "first-time" N-grams is just the number of N-gram *types* we saw in the data (since we had to see each type for the first time exactly once).

So we estimate the *total* probability mass of all the zero N-grams with the number of types divided by the number of tokens plus observed types:

$$\sum_{i:c_i=0} p_i^* = \frac{T}{N+T} \tag{6.16}$$

Why do we normalize by the number of tokens plus types? We can think of our training corpus as a series of events; one event for each token and one event for each new type. So Equation 6.16 gives the Maximum Likelihood Estimate of the probability of a new type event occurring. Note that the number of observed types T is different than the "total types" or "vocabulary size V" that we used in add-one smoothing: T is the types we have already seen, while V is the total number of possible types we might ever see.

Equation 6.16 gives the total "probability of unseen N-grams". We need to divide this up among all the zero N-grams. We could just choose to divide it equally. Let Z be the total number of N-grams with count zero (types; there aren't any tokens). Each formerly-zero unigram now gets its equal share of the redistributed probability mass:

$$Z = \sum_{i:c_i=0} 1 \tag{6.17}$$

$$p_i^* = \frac{T}{Z(N+T)} \tag{6.18}$$

If the total probability of zero N-grams is computed from Equation (6.16), the extra probability mass must come from somewhere; we get it by discounting the probability of all the seen N-grams as follows:

$$p_i^* = \frac{c_i}{N+T} \text{if } (c_i > 0) \tag{6.19}$$

Alternatively, we can represent the smoothed counts directly as:

$$c_i^* = \begin{cases} \frac{T}{Z}\frac{N}{N+T}, & \text{if } c_i = 0 \\ c_i\frac{N}{N+T}, & \text{if } c_i > 0 \end{cases} \tag{6.20}$$

Witten-Bell discounting looks a lot like add-one smoothing for unigrams. But if we extend the equation to bigrams we will see a big difference. This is because now our type-counts are conditioned on some history. In order to compute the probability of a bigram $w_{n-1}w_{n-2}$ we haven't seen, we use "the probability of seeing a new bigram starting with w_{n-1}". This lets our estimate of "first-time bigrams" be specific to a word history. Words that tend to occur in a smaller number of bigrams will supply a lower "unseen-bigram" estimate than words that are more promiscuous.

We represent this fact by conditioning T, the number of bigram types, and N, the number of bigram tokens, on the previous word w_x , as follows:

$$\sum_{i:c(w_x w_i)=0} p^*(w_i|w_x) = \frac{T(w_x)}{N(w_x)+T(w_x)} \tag{6.21}$$

Again, we will need to distribute this probability mass among all the unseen bigrams. Let Z again be the total number of bigrams with a given first word that have count zero (types; there aren't any tokens). Each formerly zero bigram now gets its equal share of the redistributed probability mass:

$$Z(w_x) = \sum_{i:c(w_x w_i)=0} 1 \tag{6.22}$$

$$p^*(w_i|w_{i-1}) = \frac{T(w_{i-1})}{Z(w_{i-1})(N+T(w_{i-1}))} \text{ if } (c_{w_{i-1}w_i} = 0) \tag{6.23}$$

As for the non-zero bigrams, we discount them in the same manner, by parameterizing T on the history:

$$\sum_{i:c(w_x w_i)>0} p^*(w_i|w_x) = \frac{c(w_x w_i)}{c(w_x)+T(w_x)} \tag{6.24}$$

To use Equation 6.24 to smooth the restaurant bigram from Figure 6.5, we will need the number of bigram types $T(w)$ for each of the first words. Here are those values:

I	95
want	76
to	130
eat	124
Chinese	20
food	82
lunch	45

In addition we will need the Z values for each of these words. Since we know how many words we have in the vocabulary ($V = 1{,}616$), there are exactly V possible bigrams that begin with a given word w, so the number of unseen bigram types with a given prefix is V minus the number of observed types:

$$Z(w) = V - T(w) \tag{6.25}$$

Here are those Z values:

I	1,521
want	1,540
to	1,486
eat	1,492
Chinese	1,596
food	1,534
lunch	1,571

Figure 6.9 shows the discounted restaurant bigram counts.

	I	want	to	eat	Chinese	food	lunch
I	8	1060	.062	13	.062	.062	.062
want	3	.046	740	.046	6	8	6
to	3	.085	10	827	3	.085	12
eat	.075	.075	2	.075	17	2	46
Chinese	2	.012	.012	.012	.012	109	1
food	18	.059	16	.059	.059	.059	.059
lunch	4	.026	.026	.026	.026	1	.026

Figure 6.9 Witten-Bell smoothed bigram counts for seven of the words (out of 1616 total word types) in the Berkeley Restaurant Project corpus of ≈10,000 sentences.

The discount values for the Witten-Bell algorithm are much more reasonable than for add-one smoothing:

I	.97
want	.94
to	.96
eat	.88
Chinese	.91
food	.94
lunch	.91

It is also possible to use Witten-Bell (or other) discounting in a different way. In Equation (6.21), we conditioned the smoothed bigram probabilities on the previous word. That is, we conditioned the number of types $T(w_x)$ and tokens $N(w_x)$ on the previous word w_x. But we could choose instead to treat a bigram as if it were a single event, ignoring the fact that it is composed of two words. Then T would be the number of types of *all* bigrams, and N would be the number of tokens of *all* bigrams that occurred. Treating the bigrams as a unit in this way, we are essentially discounting, not the conditional probability $P(w_i|w_x)$, but the **joint probability** $P(w_x w_i)$. In this way the probability $P(w_x w_i)$ is treated just like a unigram probability. This kind of discounting is less commonly used than the "conditional" discounting we walked through above starting with Equation 6.21. (Although it is often used for the Good-Turing discounting algorithm described below).

JOINT PROBABILITY

In Section 6.4 we show that discounting also plays a role in more sophisticated language models. Witten-Bell discounting is commonly used in speech recognition systems such as Placeway et al. (1993).

Good-Turing Discounting

This section introduces a slightly more complex form of discounting than the Witten-Bell algorithm called **Good-Turing** smoothing. This section may be skipped by readers who are not focusing on discounting algorithms.

GOOD-TURING

The Good-Turing algorithm was first described by Good (1953), who credits Turing with the original idea; a complete proof is presented in Church et al. (1991). The basic insight of Good-Turing smoothing is to re-estimate the amount of probability mass to assign to N-grams with zero or low counts by looking at the number of N-grams with higher counts. In other words, we examine N_c, the number of N-grams that occur c times. We refer to the number of N-grams that occur c times as the frequency of frequency c. So applying the idea to smoothing the joint probability of bigrams, N_0 is the

number of bigrams b of count 0, N_1 the number of bigrams with count 1, and so on:

$$N_c = \sum_{b:c(b)=c} 1 \tag{6.26}$$

The Good-Turing estimate gives a smoothed count c^* based on the set of N_c for all c, as follows:

$$c^* = (c+1)\frac{N_{c+1}}{N_c} \tag{6.27}$$

For example, the revised count for the bigrams that never occurred (c_0) is estimating by dividing the number of bigrams that occurred once (the **singleton** or **hapax legomenon** bigrams N_1) by the number of bigrams that never occurred (N_0). Using the count of things we've seen once to estimate the count of things we've never seen should remind you of the Witten-Bell discounting algorithm we saw earlier in this chapter. The Good-Turing algorithm was first applied to the smoothing of N-gram grammars by Katz, as cited in Nádas (1984). Figure 6.10 gives an example of the application of Good-Turing discounting to a bigram grammar computed by Church and Gale (1991) from 22 million words from the Associated Press (AP) newswire. The first column shows the count c, i.e., the number of observed instances of a bigram. The second column shows the number of bigrams that had this count. Thus 449,721 bigrams has a count of 2. The third column shows c^*, the Good-Turing re-estimation of the count.

SINGLETON

c (MLE)	N_c	c^* (GT)
0	74,671,100,000	0.0000270
1	2,018,046	0.446
2	449,721	1.26
3	188,933	2.24
4	105,668	3.24
5	68,379	4.22
6	48,190	5.19
7	35,709	6.21
8	27,710	7.24
9	22,280	8.25

Figure 6.10 Bigram "frequencies of frequencies" from 22 million AP bigrams, and Good-Turing re-estimations after Church and Gale (1991).

Church et al. (1991) show that the Good-Turing estimate relies on the assumption that the distribution of each bigram is binomial. The estimate

also assumes we know N_0, the number of bigrams we haven't seen. We know this because given a vocabulary size of V, the total number of bigrams is V^2. (N_0 is V^2 minus all the bigrams we have seen).

In practice, this discounted estimate c^* is not used for all counts c. Large counts (where $c > k$ for some threshold k) are assumed to be reliable. Katz (1987) suggests setting k at 5. Thus we define

$$c^* = c \text{ for } c > k \tag{6.28}$$

The correct equation for c^* when some k is introduced (from Katz (1987)) is:

$$c^* = \frac{(c+1)\frac{N_{c+1}}{N_c} - c\frac{(k+1)N_{k+1}}{N_1}}{1 - \frac{(k+1)N_{k+1}}{N_1}}, \text{ for } 1 \leq c \leq k. \tag{6.29}$$

With Good-Turing discounting as with any other, it is usual to treat N-grams with low counts (especially counts of 1) as if the count was 0.

6.4 BACKOFF

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$ to help us compute $P(w_n|w_{n-1}w_{n-2})$, we can estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$.

DELETED INTERPOLATION
BACKOFF

There are two ways to rely on this N-gram "hierarchy", **deleted interpolation** and **backoff**. We will focus on backoff, although we give a quick overview of deleted interpolation after this section. Backoff N-gram modeling is a nonlinear method introduced by Katz (1987). In the backoff model, like the deleted interpolation model, we build an N-gram model based on an $(N-1)$-gram model. The difference is that in backoff, if we have non-zero trigram counts, we rely solely on the trigram counts and don't interpolate the bigram and unigram counts at all. We only "back off" to a lower order N-gram if we have zero evidence for a higher-order N-gram.

The trigram version of backoff might be represented as follows:

$$\hat{P}(w_i|w_{i-2}w_{i-1}) = \begin{cases} P(w_i|w_{i-2}w_{i-1}), & \text{if } C(w_{i-2}w_{i-1}w_i) > 0 \\ \alpha_1 P(w_i|w_{i-1}), & \text{if } C(w_{i-2}w_{i-1}w_i) = 0 \\ & \text{and } C(w_{i-1}w_i) > 0 \\ \alpha_2 P(w_i), & \text{otherwise.} \end{cases} \tag{6.30}$$

Let's ignore the α values for a moment; we'll discuss the need for these weighting factors below. Here's a first pass at the (recursive) equation for representing the general case of this form of backoff.

$$\begin{aligned}\hat{P}(w_n|w_{n-N+1}^{n-1}) &= \tilde{P}(w_n|w_{n-N+1}^{n-1}) \\ &\quad + \theta(P(w_n|w_{n-N+1}^{n-1}))\alpha\hat{P}(w_n|w_{n-N+2}^{n-1})\end{aligned} \tag{6.31}$$

Again, ignore the α and the $\tilde{P}$ for the moment. Following Katz, we've used θ to indicate the binary function that selects a lower ordered model only if the higher-order model gives a zero probability:

$$\theta(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{otherwise.} \end{cases} \tag{6.32}$$

and each $P(\cdot)$ is a MLE (i.e., computed directly by dividing counts). The next section will work through these equations in more detail. In order to do that, we'll need to understand the role of the α values and how to compute them.

Combining Backoff with Discounting

Our previous discussions of discounting showed how to use a discounting algorithm to assign probability mass to unseen events. For simplicity, we assumed that these unseen events were all equally probable, and so the probability mass got distributed evenly among all unseen events. Now we can combine discounting with the backoff algorithm we have just seen to be a little more clever in assigning probability to unseen events. We will use the discounting algorithm to tells us how much total probability mass to set aside for all the events we haven't seen, and the backoff algorithm to tell us how to distribute this probability in a clever way.

First, the reader should stop and answer the following question (don't look ahead): Why did we need the α values in Equation (6.30) (or Equation (6.31))? Why couldn't we just have three sets of probabilities without weights?

The answer: without α values, the result of the equation would not be a true probability! This is because the original $P(w_n|w_{n-N+1}^{n-1})$ we got from relative frequencies were true probabilities, that is, if we sum the probability of a given w_n over all N-gram contexts, we should get 1:

$$\sum_{i,j} P(w_n|w_i w_j) = 1 \tag{6.33}$$

But if that is the case, if we back off to a lower order model when the probability is zero, we are adding extra probability mass into the equation, and the total probability of a word will be greater than 1!

Thus any backoff language model must also be discounted. This explains the αs and $\tilde{P}$ in Equation 6.31. The $\tilde{P}$ comes from our need to discount the MLE probabilities to save some probability mass for the lower order N-grams. We will use $\tilde{P}$ to mean discounted probabilities, and save P for plain old relative frequencies computed directly from counts. The α is used to ensure that the probability mass from all the lower order N-grams sums up to exactly the amount that we saved by discounting the higher-order N-grams. Here's the correct final equation:

$\tilde{P}$

$$\begin{aligned}\hat{P}(w_n|w_{n-N+1}^{n-1}) &= \tilde{P}(w_n|w_{n-N+1}^{n-1}) \\ &\quad + \theta(P(w_n|w_{n-N+1}^{n-1})) \\ &\quad \cdot \alpha(w_{n-N+1}^{n-1})\hat{P}(w_n|w_{n-N+2}^{n-1}) \end{aligned} \tag{6.34}$$

Now let's see the formal definition of each of these components of the equation. We define $\tilde{P}$ as the discounted (c^*) MLE estimate of the conditional probability of an N-gram, as follows:

$$\tilde{P}(w_n|w_{n-N+1}^{n-1}) = \frac{c^*(w_{n-N+1}^{n})}{c(w_1^{n-N+1})} \tag{6.35}$$

This probability $\tilde{P}$ will be slightly less than the MLE estimate

$$\frac{c(w_{n-N+1}^{n})}{c(w_{n-N+1}^{n-1})}$$

(i.e., on average the c^* will be less than c). This will leave some probability mass for the lower order N-grams. Now we need to build the α weighting we'll need for passing this mass to the lower order N-grams. Let's represent the total amount of left-over probability mass by the function β, a function of the $N-1$-gram context. For a given $N-1$-gram context, the total left-over probability mass can be computed by subtracting from 1 the total discounted probability mass for all N-grams starting with that context:

$$\beta(w_{n-N+1}^{n-1}) = 1 - \sum_{w_n:c(w_{n-N+1}^{n})>0} \tilde{P}(w_n|w_{n-N+1}^{n-1}) \tag{6.36}$$

This gives us the total probability mass that we are ready to distribute to all $N-1$-gram (e.g., bigrams if our original model was a trigram). Each individual $N-1$-gram (bigram) will only get a fraction of this mass, so we need to normalize β by the total probability of all the $N-1$-grams (bigrams)

that begin some N-gram (trigram). The final equation for computing how much probability mass to distribute from an N-gram to an $N-1$-gram is represented by the function α:

$$\alpha(w_{n-N+1}^{n-1}) = \frac{1 - \sum_{w_n : c(w_{n-N+1}^{n}) > 0} \tilde{P}(w_n | w_{n-N+1}^{n-1})}{1 - \sum_{w_n : c(w_{n-N+1}^{n}) > 0} \tilde{P}(w_n | w_{n-N+2}^{n-1})} \tag{6.37}$$

Note that α is a function of the preceding word string, that is, of w_{n-N+1}^{n-1}; thus the amount by which we discount each trigram (d), and the mass that gets reassigned to lower order N-grams (α) are recomputed for every N-gram (more accurately for every $N-1$-gram that occurs in any N-gram).

We only need to specify what to do when the counts of an $N-1$-gram context are 0, (i.e., when $c(w_{n-N+1}^{m-1}) = 0$) and our definition is complete:

$$P(w_n | w_{n-N+1}^{n-N+1}) = P(w_n | w_{n-N+1}^{n-N+2}) \tag{6.38}$$

and

$$\tilde{P}(w_n | w_{n-N+1}^{n-1}) = 0 \tag{6.39}$$

and

$$\tilde{\beta}(w_{n-N+1}^{n-1}) = 1 \tag{6.40}$$

In Equation (6.35), the discounted probability $\tilde{P}$ can be computed with the discounted counts c^* from the Witten-Bell discounting (Equation (6.20)) or with the Good-Turing discounting discussed below.

Here is the backoff model expressed in a slightly clearer format in its trigram version:

$$\hat{P}(w_i | w_{i-2} w_{i-1}) = \begin{cases} \tilde{P}(w_i | w_{i-2} w_{i-1}), & \text{if } C(w_{i-2} w_{i-1} w_i) > 0 \\ \alpha(w_{n-2}^{n-1}) \tilde{P}(w_i | w_{i-1}), & \text{if } C(w_{i-2} w_{i-1} w_i) = 0 \\ & \text{and } C(w_{i-1} w_i) > 0 \\ \alpha(w_{n-1}) \tilde{P}(w_i), & \text{otherwise.} \end{cases}$$

In practice, when discounting, we usually ignore counts of 1, that is, we treat N-grams with a count of 1 as if they never occurred.

Gupta et al. (1992) present a variant backoff method of assigning probabilities to zero trigrams.

6.5 DELETED INTERPOLATION

The deleted interpolation algorithm, due to Jelinek and Mercer (1980), combines different N-gram orders by linearly interpolating all three models whenever we are computing any trigram. That is, we estimate the probability $P(w_n|w_{n-1}w_{n-2})$ by mixing together the unigram, bigram, and trigram probabilities. Each of these is weighted by a linear weight λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-1}w_{n-2}) &= \lambda_1 P(w_n|w_{n-1}w_{n-2}) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n)\end{aligned} \tag{6.41}$$

such that the λs sum to 1:

$$\sum_i \lambda_i = 1 \tag{6.42}$$

DELETED INTERPOLATION

In practice, in this deleted interpolation **deleted interpolation** algorithm we don't train just three λs for a trigram grammar. Instead, we make each λ a function of the context. This way if we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, and so we can make the lambdas for those trigrams higher and thus give that trigram more weight in the interpolation. So a more detailed version of the interpolation formula would be:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1(w_{n-2}^{n-1})P(w_n|w_{n-2}w_{n-1}) \\ &\quad +\lambda_2(w_{n-2}^{n-1})P(w_n|w_{n-1}) \\ &\quad +\lambda_3(w_{n-2}^{n-1})P(w_n)\end{aligned} \tag{6.43}$$

Given the $P(w_{...})$ values, the λ values are trained so as to maximize the likelihood of a *held-out* corpus separate from the main training corpus, using a version of the **EM** algorithm defined in Chapter 7 (Baum, 1972; Dempster et al., 1977; Jelinek and Mercer, 1980). Further details of the algorithm are described in Bahl et al. (1983).

6.6 N-GRAMS FOR SPELLING AND PRONUNCIATION

In Chapter 5 we saw the use of the Bayesian/noisy-channel algorithm for correcting spelling errors and for picking a word given a surface pronunci-

ation. We saw that both these algorithms failed, returning the wrong word, because they had no way to model the probability of multiple-word strings. Now that our n-grams give us such a model, we return to these two problems.

Context-Sensitive Spelling Error Correction

Chapter 5 introduced the idea of detecting spelling errors by looking for words that are not in a dictionary, are not generated by some finite-state model of English word-formation, or have low probability orthotactics. But none of these techniques is sufficient to detect and correct **real-word** spelling errors. **real-word error detection**. This is the class of errors that result in an actual word of English. This can happen from typographical errors (insertion, deletion, transposition) that accidently produce a real word (e.g., *there* for *three*), or because the writer substituted the wrong spelling of a homophone or near-homophone (e.g., *dessert* for *desert*, or *piece* for *peace*). The task of correcting these errors is called **context-sensitive spelling error correction**.

REAL-WORD ERROR DETECTION

How important are these errors? By an a priori analysis of single typographical errors (single insertions, deletions, substitutions, or transpositions) Peterson (1986) estimates that 15% of such spelling errors produce valid English words (given a very large list of 350,000 words). Kukich (1992) summarizes a number of other analyses based on empirical studies of corpora, which give figures between of 25% and 40% for the percentage of errors that are valid English words. Figure 6.11 gives some examples from Kukich (1992), broken down into **local** and **global** errors. Local errors are those that are probably detectable from the immediate surrounding words, while global errors are ones in which error detection requires examination of a large context.

One method for context-sensitive spelling error correction is based on N-grams.

The word N-gram approach to spelling error detection and correction was proposed by Mays et al. (1991). The idea is to generate every possible misspelling of each word in a sentence either just by typographical modifications (letter insertion, deletion, substitution), or by including homophones as well, (and presumably including the correct spelling), and then choosing the spelling that gives the sentence the highest prior probability. That is, given a sentence $W = \{w_1, w_2, \ldots, w_k, \ldots, w_n\}$, where w_k has alternative spelling w'_k, w''_k, etc., we choose the spelling among these possible spellings that maximizes $P(W)$, using the N-gram grammar to compute $P(W)$. A

Local Errors
The study was conducted mainly *be* John Black.
They are leaving in about fifteen *minuets* to go to her house.
The design *an* construction of the system will take more than a year.
Hopefully, all *with* continue smoothly in my absence.
Can they *lave* him my messages?
I need to *notified* the bank of [this problem.]
He *need* to go there right *no w*.
He is trying to *fine* out.
Global Errors
Won't they *heave if* next Monday at that time?
This thesis is supported by the fact that since 1989 the system has been operating *system* with all four units on-line, but ...

Figure 6.11 Some attested real-word spelling errors from Kukich (1992), broken down into **local** and **global** errors.

class-based N-gram can be used instead, which can find unlikely part-of-speech combinations, although it may not do as well at to finding unlikely word combinations.

There are many other statistical approaches to context-sensitive spelling error correction, some proposed directly for spelling, other for more general types of lexical disambiguation (such as word-sense disambiguation or accent restoration). Beside the trigram approach we have just described, these include Bayesian classifiers, alone or combined with trigrams (Gale et al., 1993; Golding, 1997; Golding and Schabes, 1996), decision lists (Yarowsky, 1994), transformation based learning (Mangu and Brill, 1997), latent semantic analysis (Jones and Martin, 1997), and Winnow (Golding and Roth, 1999). In a comparison of these, Golding and Roth (1999) found the Winnow algorithm gave the best performance. In general, however, these algorithms are very similar in many ways; they are all based on features like word and part-of-speech N-grams, and Roth (1998, 1999) shows that many of them make their predictions using a family of linear predictors called **Linear Statistical Queries (LSQ) hypotheses**. Chapter 17 will define all these algorithms and discuss these issues further in the context of word-sense disambiguation.

N-grams for Pronunciation Modeling

The N-gram model can also be used to get better performance on the words-from-pronunciation task that we studied in Chapter 5. Recall that the input was the pronunciation [n iy] following the word *I*. We said that the five words that could be pronounced [n iy] were *need*, *new*, *neat*, *the*, and *knee*. The algorithm in Chapter 5 was based on the product of the unigram probability of each word and the pronunciation likelihood, and incorrectly chose the word *new*, based mainly on its high unigram probability.

Adding a simple bigram probability, even without proper smoothing, is enough to solve this problem correctly. In the following table we fix the table on page 167 by using a bigram rather than unigram word probability $p(w)$ for each of the five candidate words (given that the word *I* occurs 64,736 times in the combined Brown and Switchboard corpora):

Word	C('I' w)	C('I' w)+0.5	p(w|'I')
need	153	153.5	.0016
new	0	0.5	.000005
knee	0	0.5	.000005
the	17	17.5	.00018
neat	0	0.5	.000005

Incorporating this new word probability into combined model, it now predicts the correct word *need*, as the table below shows:

Word	p(y|w)	p(w)	p(y|w)p(w)
need	.11	.0016	.00018
knee	1.00	.000005	.000005
neat	.52	.000005	.0000026
new	.36	.000005	.0000018
the	0	.00018	0

6.7 ENTROPY

I got the horse right here
Frank Loesser, Guys and Dolls

Entropy and **perplexity** are the most common metrics used to evaluate N-gram systems. The next sections summarize a few necessary fundamental facts about **information theory** and then introduce the entropy and perplexity metrics. We strongly suggest that the interested reader consult a good

information theory textbook; Cover and Thomas (1991) is one excellent example.

ENTROPY

Entropy is a measure of information, and is invaluable in natural language processing, speech recognition, and computational linguistics. It can be used as a metric for how much information there is in a particular grammar, for how well a given grammar matches a given language, for how predictive a given N-gram grammar is about what the next word could be. Given two grammars and a corpus, we can use entropy to tell us which grammar better matches the corpus. We can also use entropy to compare how difficult two speech recognition tasks are, and also to measure how well a given probabilistic grammar matches human grammars.

Computing entropy requires that we establish a random variable X that ranges over whatever we are predicting (words, letters, parts of speech, the set of which we'll call χ), and that has a particular probability function, call it $p(x)$. The entropy of this random variable X is then

$$H(X) = -\sum_{x \in \chi} p(x) \log_2 p(x) \tag{6.44}$$

The log can in principle be computed in any base; recall that we use log base 2 in all calculations in this book. The result of this is that the entropy is measured in **bits**.

The most intuitive way to define entropy for computer scientists is to think of the entropy as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme.

Cover and Thomas (1991) suggest the following example. Imagine that we want to place a bet on a horse race but it is too far to go all the way to Yonkers Racetrack, and we'd like to send a short message to the bookie to tell him which horse to bet on. Suppose there are eight horses in this particular race.

One way to encode this message is just to use the binary representation of the horse's number as the code; thus horse 1 would be `001`, horse 2 `010`, horse 3 `011`, and so on, with horse 8 coded as `000`. If we spend the whole day betting, and each horse is coded with 3 bits, on the average we would be sending 3 bits per race.

Can we do better? Suppose that the spread is the actual distribution of the bets placed, and that we represent it as the prior probability of each horse as follows:

Horse 1	$\frac{1}{2}$	Horse 5	$\frac{1}{64}$
Horse 2	$\frac{1}{4}$	Horse 6	$\frac{1}{64}$
Horse 3	$\frac{1}{8}$	Horse 7	$\frac{1}{64}$
Horse 4	$\frac{1}{16}$	Horse 8	$\frac{1}{64}$

The entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits, and is:

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i)\log p(i) \\ &= -\tfrac{1}{2}\log\tfrac{1}{2} - \tfrac{1}{4}\log\tfrac{1}{4} - \tfrac{1}{8}\log\tfrac{1}{8} - \tfrac{1}{16}\log\tfrac{1}{16} - 4(\tfrac{1}{64}\log\tfrac{1}{64}) \\ &= 2 \text{ bits} \end{aligned} \tag{6.45}$$

A code that averages 2 bits per race can be built by using short encodings for more probable horses, and longer encodings for less probable horses. For example, we could encode the most likely horse with the code `0`, and the remaining horses as `10`, then `110`, `1110`, `111100`, `111101`, `111110`, and `111111`.

What if the horses are equally likely? We saw above that if we use an equal-length binary code for the horse numbers, each horse took 3 bits to code, and so the average was 3. Is the entropy the same? In this case each horse would have a probability of $\frac{1}{8}$. The entropy of the choice of horses is then:

$$H(X) = -\sum_{i=1}^{i=8} \frac{1}{8}\log\frac{1}{8} = -\log\frac{1}{8} = 3 \text{ bits} \tag{6.46}$$

The value 2^H is called the **perplexity** (Jelinek et al., 1977; Bahl et al., 1983). Perplexity can be intuitively thought of as the weighted average number of choices a random variable has to make. Thus choosing between 8 equally likely horses (where $H = 3$ bits), the perplexity is 2^3 or 8. Choosing between the biased horses in the table above (where $H = 2$ bits), the perplexity is 2^2 or 4. PERPLEXITY

Until now we have been computing the entropy of a single variable. But most of what we will use entropy for involves *sequences*; for a grammar, for example, we will be computing the entropy of some sequence of words $W = \{\ldots w_0, w_1, w_2, \ldots, w_n\}$. One way to do this is to have a variable that ranges over sequences of words. For example we can compute the entropy of a random variable that ranges over all finite sequences of words of length

b in some language L as follows:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \tag{6.47}$$

ENTROPY RATE

We could define the **entropy rate** (we could also think of this as the **per-word entropy**) as the entropy of this sequence divided by the number of words:

$$\frac{1}{n} H(W_1^n) = -\frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \tag{6.48}$$

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, its entropy rate $H(L)$ is defined as:

$$\begin{aligned} H(L) &= \lim_{n \to \infty} \frac{1}{n} H(w_1, w_2, \ldots, w_n) \\ &= \lim_{n \to \infty} \frac{1}{n} \sum_{W \in L} p(w_1, \ldots, w_n) \log p(w_1, \ldots, w_n) \end{aligned} \tag{6.49}$$

The Shannon-McMillan-Breiman theorem (Algoet and Cover, 1988; Cover and Thomas, 1991) states that if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

$$H(L) = \lim_{n \to \infty} -\frac{1}{n} \log p(w_1 w_2 \ldots w_n) \tag{6.50}$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman theorem is that a long enough sequence of words will contain in it many other shorter sequences, and that each of these shorter sequences will reoccur in the longer sequence according to their probabilities.

STATIONARY

A stochastic process is said to be **stationary** if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time $t+1$. Markov models, and hence N-grams, are stationary. For example, in a bigram, P_i is dependent only on P_{i-1}. So if we shift our time index by x, P_{i+x} is still dependent on P_{i+x-1}. But natural language is not stationary, since as we will see in Chapter 9, the probability of upcoming words can be dependent on events that were arbitrarily distant and time dependent. Thus our statistical models only give an approximation to the correct distributions and entropies of natural language.

To summarize, by making some incorrect but convenient simplifying assumptions, we can compute the entropy of some stochastic process by tak-

ing a very long sample of the output, and computing its average log probability. In the next section we talk about the why and how; *why* we would want to do this (i.e., for what kinds of problems would the entropy tell us something useful), and *how* to compute the probability of a very long sequence.

Cross Entropy for Comparing Models

CROSS ENTROPY

In this section we introduce the **cross entropy**, and discuss its usefulness in comparing different probabilistic models. The cross entropy is useful when we don't know the actual probability distribution p that generated some data. It allows us to use some m, which is a model of p (i.e., an approximation to p. The cross-entropy of m on p is defined by:

$$H(p,m) = \lim_{n\to\infty} \frac{1}{n} \sum_{W\in L} p(w_1,\ldots,w_n) \log m(w_1,\ldots,w_n) \tag{6.51}$$

That is we draw sequences according to the probability distribution p, but sum the log of their probability according to m.

Again, following the Shannon-McMillan-Breiman theorem, for a stationary ergodic process:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \log m(w_1 w_2 \ldots w_n) \tag{6.52}$$

What makes the cross entropy useful is that the cross entropy $H(p,m)$ is an upper bound on the entropy $H(p)$. For any model m:

$$H(p) \leq H(p,m) \tag{6.53}$$

This means that we can use some simplified model m to help estimate the true entropy of a sequence of symbols drawn according to probability p. The more accurate m is, the closer the cross entropy $H(p,m)$ will be to the true entropy $H(p)$. Thus the difference between $H(p,m)$ and $H(p)$ is a measure of how accurate a model is. Between two models m_1 and m_2, the more accurate model will be the one with the lower cross-entropy. (The cross-entropy can never be lower than the true entropy, so a model cannot err by underestimating the true entropy).

The Entropy of English

As we suggested in the previous section, the cross-entropy of some model m can be used as an upper bound on the true entropy of some process. We can use this method to get an estimate of the true entropy of English. Why should we care about the entropy of English?

METHODOLOGY BOX: PERPLEXITY

The methodology box on page 204 mentioned the idea of computing the **perplexity of a test set** as a way of comparing two probabilistic models. (Despite the risk of ambiguity, we will follow the speech and language processing literature in using the term "perplexity" rather than the more technically correct term "cross-perplexity".) Here's an example of perplexity computation as part of a "business news dictation system". We trained unigram, bigram, and trigram Katz-style backoff grammars with Good-Turing discounting on 38 million words (including start-of-sentence tokens) from the Wall Street Journal (from the WSJ0 corpus (LDC, 1993)). We used a vocabulary of 19,979 words (i.e., the rest of the words types were mapped to the unknown word token <UNK> in both training and testing). We then computed the perplexity of each of these models on a test set of 1.5 million words (where the perplexity is defined as $2^{H(p,m)}$). The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars.

N-gram Order	Perplexity
Unigram	962
Bigram	170
Trigram	109

In computing perplexities the model m must be constructed without any knowledge of the test set t. Any kind of knowledge of the test set can cause the perplexity to be artificially low. For example, sometimes instead of mapping all unknown words to the <UNK> token, we use a **closed-vocabulary** test set in which we know in advance what the set of words is. This can greatly reduce the perplexity. As long as this knowledge is provided equally to each of the models we are comparing, the closed-vocabulary perplexity is still a useful metric for comparing models. But this cross-perplexity is no longer guaranteed to be greater than the true perplexity of the test set, and so great care must be taken in interpreting the results. In general, the perplexity of two language models is only comparable if they use the same vocabulary.

One reason is that the true entropy of English would give us a solid lower bound for all of our future experiments on probabilistic grammars. Another is that we can use the entropy values for English to help understand what parts of a language provide the most information (for example, is the predictability of English mainly based on word order, on semantics, on morphology, on constituency, or on pragmatic cues?) This can help us immensely in knowing where to focus our language-modeling efforts.

There are two common methods for computing the entropy of English. The first was employed by Shannon (1951), as part of his groundbreaking work in defining the field of information theory. His idea was to use human subjects, and to construct a psychological experiment that requires them to guess strings of letters; by looking at how many guesses it takes them to guess letters correctly we can estimate the probability of the letters, and hence the entropy of the sequence.

The actual experiment is designed as follows: we present a subject with some English text and ask the subject to guess the next letter. The subjects will use their knowledge of the language to guess the most probable letter first, the next most probable next, and so on. We record the number of guesses it takes for the subject to guess correctly. Shannon's insight was that the entropy of the number-of-guesses sequence is the same as the entropy of English. (The intuition is that given the number-of-guesses sequence, we could reconstruct the original text by choosing the "nth most probable" letter whenever the subject took n guesses). This methodology requires the use of letter guesses rather than word guesses (since the subject sometimes has to do an exhaustive search of all the possible letters!), and so Shannon computed the **per-letter entropy** of English rather than the per-word entropy. He reported an entropy of 1.3 bits (for 27 characters (26 letters plus space)). Shannon's estimate is likely to be too low, since it is based on a single text (*Jefferson the Virginian* by Dumas Malone). Shannon notes that his subjects had worse guesses (hence higher entropies) on other texts (newspaper writing, scientific work, and poetry). More recently variations on the Shannon experiments include the use of a gambling paradigm where the subjects get to bet on the next letter (Cover and King, 1978; Cover and Thomas, 1991).

The second method for computing the entropy of English helps avoid the single-text problem that confounds Shannon's results. This method is to take a very good stochastic model, train it on a very large corpus, and use it to assign a log-probability to a very long sequence of English, using the Shannon-McMillan-Breiman theorem:

$$H(\text{English}) \leq \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n) \tag{6.54}$$

For example, Brown et al. (1992) trained a trigram language model on 583 million words of English, (293,181 different types) and used it to compute the probability of the entire Brown corpus (1,014,312 tokens). The training data include newspapers, encyclopedias, novels, office correspondence, proceedings of the Canadian parliament, and other miscellaneous sources.

They then computed the character-entropy of the Brown corpus, by using their word-trigram grammar to assign probabilities to the Brown corpus, considered as a sequence of individual letters. They obtained an entropy of 1.75 bits per character (where the set of characters included all the 95 printable ASCII characters).

The average length of English written words (including space) has been reported at 5.5 letters (Nádas, 1984). If this is correct, it means that the Shannon estimate of 1.3 bits per letter corresponds to a per-word perplexity of 142 for general English. The numbers we report above for the WSJ experiments are significantly lower since the training and test set came from same subsample of English. That is, those experiments underestimate the complexity of English since the Wall Street Journal looks very little like Shakespeare.

Bibliographical and Historical Notes

The underlying mathematics of the N-gram was first proposed by Markov (1913), who used what are now called **Markov chains** (bigrams and trigrams) to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a consonant. Markov classified 20,000 letters as V or C and computed the bigram and trigram probability that a given letter would be a vowel given the previous one or two letters. Shannon (1948) applied N-grams to compute approximations to English word sequences. Based on Shannon's work, Markov models were commonly used in modeling word sequences by the 1950s. In a series of extremely influential papers starting with Chomsky (1956) and including Chomsky (1957) and Miller and Chomsky (1963), Noam Chomsky argued that "finite-state Markov processes", while a possibly useful engineering heuristic, were incapable of being a complete cognitive model of human grammatical knowledge. These arguments led many linguists and computational linguists away from statistical models altogether.

The resurgence of N-gram models came from Jelinek, Mercer, Bahl, and colleagues at the IBM Thomas J. Watson Research Center, influenced by Shannon, and Baker at CMU, influenced by the work of Baum and colleagues. These two labs independently successfully used N-grams in their speech recognition systems (Jelinek, 1976; Baker, 1975; Bahl et al., 1983). The Good-Turing algorithm was first applied to the smoothing of N-gram grammars at IBM by Katz, as cited in Nádas (1984). Jelinek (1990) summarizes this and many other early language model innovations used in the IBM language models.

While smoothing had been applied as an engineering solution to the zero-frequency problem at least as early as Jeffreys (1948) (add-one smoothing), it is only relatively recently that smoothing received serious attention. Church and Gale (1991) gives a good description of the Good-Turing method, as well as the proof, and also gives a good description of the Deleted Interpolation method and a new smoothing method. Sampson (1996) also has a useful discussion of Good-Turing. Problems with the Add-one algorithm are summarized in Gale and Church (1994). Method C in Witten and Bell (1991) describes what we called Witten-Bell discounting. Chen and Goodman (1996) give an empirical comparison of different smoothing algorithms, including two new methods, *average-count* and *one-count*, as well as Church and Gale's. Iyer and Ostendorf (1997) discuss a way of smoothing by adding in data from additional corpora.

Much recent work on language modeling has focused on ways to build more sophisticated N-grams. These approaches include giving extra weight to N-grams which have already occurred recently (the **cache LM** of Kuhn and de Mori (1990)), choosing long-distance **triggers** instead of just local N-grams (Rosenfeld, 1996; Niesler and Woodland, 1999; Zhou and Lua, 1998), and using **variable-length N-grams** (Ney et al., 1994; Kneser, 1996; Niesler and Woodland, 1996). Another class of approaches use semantic information to enrich the N-gram, including semantic word associations based on the **latent semantic indexing** described in Chapter 15 (Coccaro and Jurafsky, 1998; Bellegarda, 1999)), and from on-line dictionaries or thesauri (Demetriou et al., 1997). **Class-based** N-grams, based on word classes such as parts-of-speech, are described in Chapter 8. Language models based on more structured linguistic knowledge (such as probabilistic parsers) are described in Chapter 12. Finally, a number of augmentations to N-grams are based on discourse knowledge, such as using knowledge of the current topic (Chen et al., 1998; Seymore and Rosenfeld, 1997; Seymore et al., 1998; Florian and Yarowsky, 1999; Khudanpur and Wu, 1999) or the current speech act in dialogue (see Chapter 19).

CACHE LM

TRIGGERS

VARIABLE-LENGTH N-GRAMS

LATENT SEMANTIC INDEXING

CLASS-BASED

6.8 SUMMARY

This chapter introduced the N-gram, one of the oldest and most broadly useful practical tools in language processing.

- An N-gram probability is the conditional probability of a word given the previous $N-1$ words. N-gram probabilities can be computed by simply counting in a corpus and normalizing (the **Maximum Likelihood Estimate**) or they can be computed by more sophisticated algorithms. The advantage of N-grams is that they take advantage of lots of rich lexical knowledge. A disadvantage for some purposes is that they are very dependent on the corpus they were trained on.
- **Smoothing** algorithms provide a better way of estimating the probability of N-grams which never occur. Commonly-used smoothing algorithms include **backoff** or **deleted interpolation**, with **Witten-Bell** or **Good-Turing** discounting.
- Corpus-based **language models** like N-grams are evaluated by separating the corpus into a **training set** and a **test set**, training the model on the training set, and evaluating on the test set. The **entropy** H, or more commonly the **perplexity** 2^H (more properly **cross-entropy** and **cross-perplexity**) of a test set are used to compare language models.

EXERCISES

6.1 Write out the equation for trigram probability estimation (modifying Equation 6.11).

6.2 Write out the equation for the discount $d = \frac{c*}{c}$ for add-one smoothing. Do the same for Witten-Bell smoothing. How do they differ?

6.3 Write a program (Perl is sufficient) to compute unsmoothed unigrams and bigrams.

6.4 Run your N-gram program on two different small corpora of your choice (you might use email text or newsgroups). Now compare the statistics

of the two corpora. What are the differences in the most common unigrams between the two? How about interesting differences in bigrams?

6.5 Add an option to your program to generate random sentences.

6.6 Add an option to your program to do Witten-Bell discounting.

6.7 Add an option to your program to compute the entropy (or perplexity) of a test set.

6.8 Suppose someone took all the words in a sentence and reordered them randomly. Write a program which take as input such a **bag of words** and produces as output a guess at the original order. Use the Viterbi algorithm and an N-gram grammar produced by your N-gram program (on some corpus).

BAG OF WORDS

6.9 The field of **authorship attribution** is concerned with discovering the author of a particular text. Authorship attribution is important in many fields, including history, literature, and forensic linguistics. For example Mosteller and Wallace (1964) applied authorship identification techniques to discover who wrote *The Federalist* papers. The Federalist papers were written in 1787-1788 by Alexander Hamilton, John Jay and James Madison to persuade New York to ratify the United States Constitution. They were published anonymously, and as a result, although some of the 85 essays were clearly attributable to one author or another, the authorship of 12 were in dispute between Hamilton and Madison. Foster (1989) applied authorship identification techniques to suggest that W.S.'s *Funeral Elegy* for William Peter was probably written by William Shakespeare, and that the anonymous author of *Primary Colors* the roman à clef about the Clinton campaign for the American presidency, was journalist Joe Klein (Foster, 1996).

AUTHORSHIP ATTRIBUTION

A standard technique for authorship attribution, first used by Mosteller and Wallace, is a Bayesian approach. For example, they trained a probabilistic model of the writing of Hamilton, and another model of the writings of Madison, and computed the maximum-likelihood author for each of the disputed essays. There are many complex factors that go into these models, including vocabulary use, word-length, syllable structure, rhyme, grammar; see (Holmes, 1994) for a summary. This approach can also be used for identifying which genre a text comes from.

One factor in many models is the use of rare words. As a simple approximation to this one factor, apply the Bayesian method to the attribution of any particular text. You will need three things: a text to test, and two potential authors or genres, with a large on-line text sample of each. One of

them should be the correct author. Train a unigram language model on each of the candidate authors. You are only going to use the **singleton** unigrams in each language model. You will compute $P(T|A_1)$, the probability of the text given author or genre A_1, by (1) taking the language model from A_1, (2) by multiplying together the probabilities of all the unigrams that only occur once in the "unknown" text and (3) taking the geometric mean of these (i.e., the nth root, where n is the number of probabilities you multiplied). Do the same for A_2. Choose whichever is higher. Did it produce the correct candidate?

7 HMMS AND SPEECH RECOGNITION

When Frederic was a little lad he proved so brave and daring,
His father thought he'd 'prentice him to some career seafaring.
I was, alas! his nurs'rymaid, and so it fell to my lot
To take and bind the promising boy apprentice to a **pilot** —
A life not bad for a hardy lad, though surely not a high lot,
Though I'm a nurse, you might do worse than make your boy a pilot.
I was a stupid nurs'rymaid, on breakers always steering,
And I did not catch the word aright, through being hard of hearing;
Mistaking my instructions, which within my brain did gyrate,
I took and bound this promising boy apprentice to a **pirate.**

The Pirates of Penzance, Gilbert and Sullivan, 1877

Alas, this mistake by nurserymaid Ruth led to Frederic's long indenture as a pirate and, due to a slight complication involving 21st birthdays and leap years, nearly led to 63 extra years of apprenticeship. The mistake was quite natural, in a Gilbert-and-Sullivan sort of way; as Ruth later noted, "The two words were so much alike!" True, true; spoken language understanding is a difficult task, and it is remarkable that humans do as well at it as we do. The goal of automatic speech recognition (ASR) research is to address this problem computationally by building systems that map from an acoustic signal to a string of words. Automatic speech understanding (ASU) extends this goal to producing some sort of understanding of the sentence, rather than just the words.

The general problem of automatic transcription of speech by any speaker in any environment is still far from solved. But recent years have seen ASR technology mature to the point where it is viable in certain limited domains. One major application area is in human-computer interaction. While many tasks are better solved with visual or pointing interfaces, speech has the potential to be a better interface than the keyboard for tasks where full natural

language communication is useful, or for which keyboards are not appropriate. This includes hands-busy or eyes-busy applications, such as where the user has objects to manipulate or equipment to control. Another important application area is telephony, where speech recognition is already used for example for entering digits, recognizing "yes" to accept collect calls, or call-routing ("Accounting, please", "Prof. Regier, please"). In some applications, a multimodal interface combining speech and pointing can be more efficient than a graphical user interface without speech (Cohen et al., 1998). Finally, ASR is being applied to dictation, that is, transcription of extended monologue by a single specific speaker. Dictation is common in fields such as law and is also important as part of augmentative communication (interaction between computers and humans with some disability resulting in the inability to type, or the inability to speak). The blind Milton famously dictated *Paradise Lost* to his daughters, and Henry James dictated his later novels after a repetitive stress injury.

Different applications of speech technology necessarily place different constraints on the problem and lead to different algorithms. We chose to focus this chapter on the fundamentals of one crucial area: **Large-Vocabulary Continuous Speech Recognition (LVCSR)**, with a small section on acoustic issues in speech synthesis. Large-vocabulary generally means that the systems have a vocabulary of roughly 5,000 to 60,000 words. The term **continuous** means that the words are run together naturally; it contrasts with **isolated-word** speech recognition, in which each word must be preceded and followed by a pause. Furthermore, the algorithms we will discuss are generally **speaker-independent**; that is, they are able to recognize speech from people whose speech the system has never been exposed to before.

LVCSR

CONTINUOUS

ISOLATED-WORD

SPEAKER-INDEPENDENT

The chapter begins with an overview of speech recognition architecture, and then proceeds to introduce the HMM, the use of the Viterbi and A* algorithms for decoding, speech acoustics and features, and the use of Gaussians and MLPs to compute acoustic probabilities. Even relying on the previous three chapters, summarizing this much of the field in this chapter requires us to omit many crucial areas; the reader is encouraged to see the suggested readings at the end of the chapter for useful textbooks and articles. This chapter also includes a short section on the acoustic component of the speech synthesis algorithms discussed in Chapter 4.

7.1 Speech Recognition Architecture

Previous chapters have introduced many of the core algorithms used in speech recognition. Chapter 4 introduced the notions of **phone** and **syllable**. Chap-

ter 5 introduced the **noisy channel model**, the use of the **Bayes rule**, and the **probabilistic automaton**. Chapter 6 introduced the *N*-**gram** language model and the **perplexity** metric. In this chapter we introduce the remaining components of a modern speech recognizer: the **Hidden Markov Model** (**HMM**), the idea of **spectral features**, the **forward-backward** algorithm for HMM training, and the **Viterbi** and **stack decoding** (also called **A* decoding** algorithms for solving the **decoding** problem: mapping from strings of phone probability vectors to strings of words.

A* DECODING

Let's begin by revisiting the noisy channel model that we saw in Chapter 5. Speech recognition systems treat the acoustic input as if it were a "noisy" version of the source sentence. In order to "decode" this noisy sentence, we consider all possible sentences, and for each one we compute the probability of it generating the noisy sentence. We then chose the sentence with the maximum probability. Figure 7.1 shows this noisy-channel metaphor.

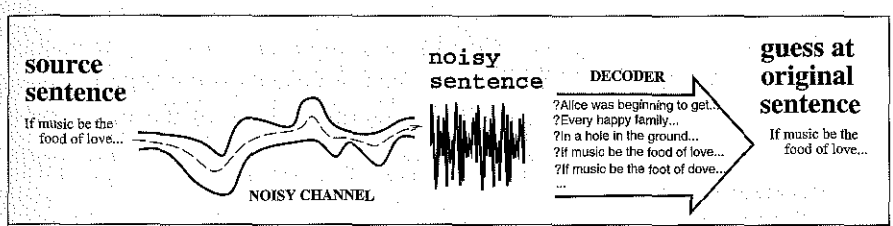


Figure 7.1 The noisy channel model applied to entire sentences (Figure 5.1 showed its application to individual words). Modern speech recognizers work by searching through a huge space of potential "source" sentences and choosing the one which has the highest probability of generating the "noisy" sentence. To do this they must have models that express the probability of sentences being realized as certain strings of words (*N*-grams), models that express the probability of words being realized as certain strings of phones (HMMs) and models that express the probability of phones being realized as acoustic or spectral features (Gaussians/MLPs).

Implementing the noisy-channel model as we have expressed it in Figure 7.1 requires solutions to two problems. First, in order to pick the sentence that best matches the noisy input we will need a complete metric for a "best match". Because speech is so variable, an acoustic input sentence will never exactly match any model we have for this sentence. As we have suggested in previous chapters, we will use probability as our metric, and will show how to combine the various probabilistic estimators to get a complete estimate for the probability of a noisy observation-sequence given a candidate

sentence. Second, since the set of all English sentences is huge, we need an efficient algorithm that will not search through all possible sentences, but only ones that have a good chance of matching the input. This is the **decoding** or **search** problem, and we will summarize two approaches: the **Viterbi** or **dynamic programming** decoder, and the **stack** or **A*** decoder.

In the rest of this introduction we will introduce the probabilistic or Bayesian model for speech recognition (or more accurately re-introduce it, since we first used the model in our discussions of spelling and pronunciation in Chapter 5); we leave discussion of decoding/search for pages 244–251.

The goal of the probabilistic noisy channel architecture for speech recognition can be summarized as follows:

> *"What is the most likely sentence out of all sentences in the language $\mathcal{L}$ given some acoustic input O?"*

We can treat the acoustic input O as a sequence of individual "symbols" or "observations" (for example by slicing up the input every 10 milliseconds, and representing each slice by floating-point values of the energy or frequencies of that slice). Each index then represents some time interval, and successive o_i indicate temporally consecutive slices of the input (note that capital letters will stand for sequences of symbols and lower-case letters for individual symbols):

$$O = o_1, o_2, o_3, \ldots, o_t \tag{7.1}$$

Similarly, we will treat a sentence as if it were composed simply of a string of words:

$$W = w_1, w_2, w_3, \ldots, w_n \tag{7.2}$$

Both of these are simplifying assumptions; for example dividing sentences into words is sometimes too fine a division (we'd like to model facts about groups of words rather than individual words) and sometimes too gross a division (we'd like to talk about morphology). Usually in speech recognition a word is defined by orthography (after mapping every word to lowercase): *oak* is treated as a different word than *oaks*, but the auxiliary *can* ("can you tell me...") is treated as the same word as the noun *can* ("i need a can of..."). Recent ASR research has begun to focus on building more sophisticated models of ASR words incorporating the morphological insights of Chapter 3 and the part-of-speech information that we will study in Chapter 8.

The probabilistic implementation of our intuition above, then, can be expressed as follows:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} P(W|O) \tag{7.3}$$

Recall that the function $\operatorname{argmax}_x f(x)$ means "the x such that f(x) is largest". Equation (7.3) is guaranteed to give us the optimal sentence W; we now need to make the equation operational. That is, for a given sentence W and acoustic sequence O we need to compute $P(W|O)$. Recall that given any probability $P(x|y)$, we can use Bayes' rule to break it down as follows:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{7.4}$$

We saw in Chapter 5 that we can substitute (7.4) into (7.3) as follows:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} \tag{7.5}$$

The probabilities on the right-hand side of (7.5) are for the most part easier to compute than $P(W|O)$. For example, $P(W)$, the prior probability of the word string itself is exactly what is estimated by the n-gram language models of Chapter 6. And we will see below that $P(O|W)$ turns out to be easy to estimate as well. But $P(O)$, the probability of the acoustic observation sequence, turns out to be harder to estimate. Luckily, we can ignore $P(O)$ just as we saw in Chapter 5. Why? Since we are maximizing over all possible sentences, we will be computing $\frac{P(O|W)P(W)}{P(O)}$ for each sentence in the language. But $P(O)$ doesn't change for each sentence! For each potential sentence we are still examining the same observations O, which must have the same probability $P(O)$. Thus:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} P(O|W)\,P(W) \tag{7.6}$$

To summarize, the most probable sentence W given some observation sequence O can be computing by taking the product of two probabilities for each sentence, and choosing the sentence for which this product is greatest. These two terms have names; $P(W)$, the **prior probability**, is called the **language model**. $P(O|W)$, the **observation likelihood**, is called the **acoustic model**.

LANGUAGE MODEL
ACOUSTIC MODEL

Key Concept #5.

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \overbrace{P(O|W)}^{\text{likelihood}} \overbrace{P(W)}^{\text{prior}} \tag{7.7}$$

We have already seen in Chapter 6 how to compute the language model prior $P(W)$ by using N-gram grammars. The rest of this chapter will show

how to compute the acoustic model $P(O|W)$, in two steps. First we will make the simplifying assumption that the input sequence is a sequence of phones F rather than a sequence of acoustic observations. Recall that we introduced the **forward** algorithm in Chapter 5, which was given "observations" that were strings of phones, and produced the probability of these phone observations given a single word. We will show that these probabilistic phone automata are really a special case of the **Hidden Markov Model**, and we will show how to extend these models to give the probability of a phone sequence given an entire sentence.

One problem with the forward algorithm as we presented it was that in order to know which word was the most-likely word (the "decoding problem"), we had to run the forward algorithm again for each word. This is clearly intractable for sentences; we can't possibly run the forward algorithm separately for each possible sentence of English. We will thus introduce two different algorithms which *simultaneously* compute the likelihood of an observation sequence given each sentence, *and* give us the most-likely sentence. These are the **Viterbi** and the **A*** decoding algorithms.

Once we have solved the likelihood-computation and decoding problems for a simplified input consisting of strings of phones, we will show how the same algorithms can be applied to true acoustic input rather than pre-defined phones. This will involve a quick introduction to acoustic input and **feature extraction**, the process of deriving meaningful features from the input soundwave. Then we will introduce the two standard models for computing phone-probabilities from these features: **Gaussian** models, and **neural net (multi-layer perceptrons)** models.

Finally, we will introduce the standard algorithm for training the Hidden Markov Models and the phone-probability estimators, the **forward-backward** or **Baum-Welch** algorithm) (Baum, 1972), a special case of the the **Expectation-Maximization** or **EM** algorithm (Dempster et al., 1977).

As a preview of the chapter, Figure 7.2 shows an outline of the components of a speech recognition system. The figure shows a speech recognition system broken down into three stages. In the **signal processing** or **feature extraction** stage, the acoustic waveform is sliced up into **frames** (usually of 10, 15, or 20 milliseconds) which are transformed into **spectral features** which give information about how much energy in the signal is at different frequencies. In the **subword** or **phone recognition** stage, we use statistical techniques like neural networks or Gaussian models to tentatively recognize individual speech sounds like *p* or *b*. For a neural network, the output of this stage is a vector of probabilities over phones for each frame (i.e., "for this

frame the probability of [p] is .8, the probability of [b] is .1, the probability of [f] is .02, etc."); for a Gaussian model the probabilities are slightly different. Finally, in the **decoding** stage, we take a dictionary of word pronunciations and a language model (probabilistic grammar) and use a Viterbi or A* **decoder** to find the sequence of words which has the highest probability given the acoustic events. DECODER

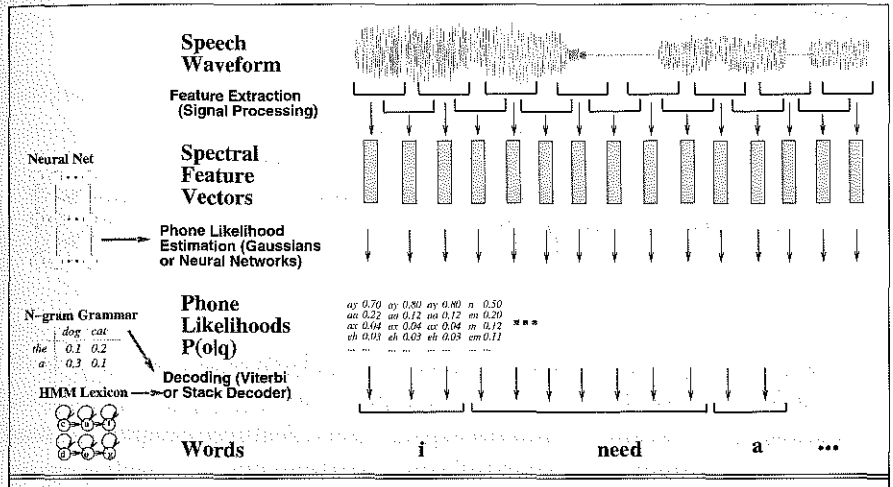


Figure 7.2 Schematic architecture for a (simplified) speech recognizer.

7.2 Overview of Hidden Markov Models

In Chapter 5 we used **weighted finite-state automata** or **Markov chains** to model the pronunciation of words. The automata consisted of a sequence of states $q = (q_0 q_1 q_2 \ldots q_n)$, each corresponding to a phone, and a set of transition probabilities between states, a_{01}, a_{12}, a_{13}, encoding the probability of one phone following another. We represented the states as nodes, and the transition probabilities as edges between nodes; an edge existed between two nodes if there was a non-zero transition probability between the two nodes. We also saw that we could use the **forward** algorithm to compute the likelihood of a sequence of observed phones $o = (o_1 o_2 o_3 \ldots o_t)$. Figure 7.3 shows an automaton for the word *need* with sample observation sequence of the kind we saw in Chapter 5.

While we will see that these models figure importantly in speech recognition, they simplify the problem in two ways. First, they assume that the

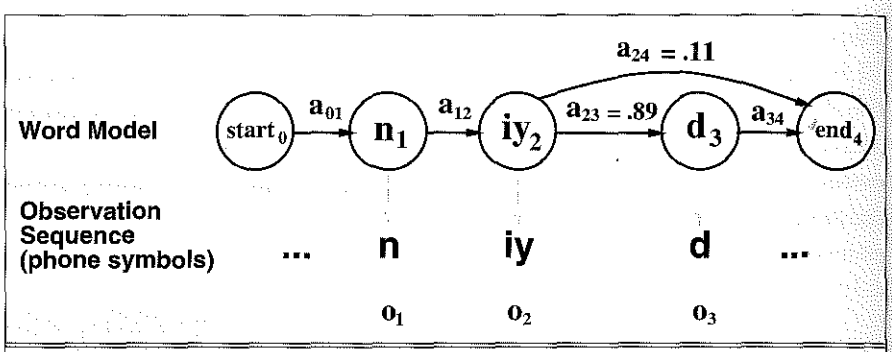


Figure 7.3 A simple weighted automaton or Markov chain pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. The transition probabilities a_{xy} between two states x and y are 1.0 unless otherwise specified.

input consists of a sequence of symbols! Obviously this is not true in the real world, where speech input consists essentially of small movements of air particles. In speech recognition, the input is an ambiguous, real-valued representation of the sliced-up input signal, called **features** or **spectral features**. We will study the details of some of these features beginning on page 259; acoustic features represent such information as how much energy there is at different frequencies. The second simplifying assumption of the weighted automata of Chapter 5 was that the input symbols correspond exactly to the states of the machine. Thus when seeing an input symbol [b], we knew that we could move into a state labeled [b]. In a **Hidden Markov Model (HMM)**, by contrast, we can't look at the input symbols and know which state to move to. The input symbols don't uniquely determine the next state.[1]

HIDDEN MARKOV MODEL

Recall that a weighted automaton or simple Markov model is specified by the set of **states** Q, the set of **transition probabilities** A, a defined **start state** and **end state(s)**, and a set of **observation likelihoods** B. For weighted automata, we defined the probabilities $b_i(o_t)$ as 1.0 if the state i matched the observation o_t and 0 if they didn't match. An HMM formally differs from a Markov model by adding two more requirements. First, it has a separate set of *observation symbols* O, which is not drawn from the same alphabet as the

[1] Actually, as we mentioned in passing, by this second criterion some of the automata we saw in Chapter 5 were technically HMMs as well. This is because the first symbol in the input string [n iy] was compatible with the [n] states in the words *need* or *an*. Seeing the symbols [n], we didn't know which underlying state it was generated by, *need-n* or *an-n*.

state set Q. Second, the observation likelihood function B is not limited to the values 1.0 and 0; in an HMM the probability $b_i(o_t)$ can take on any value from 0 to 1.0.

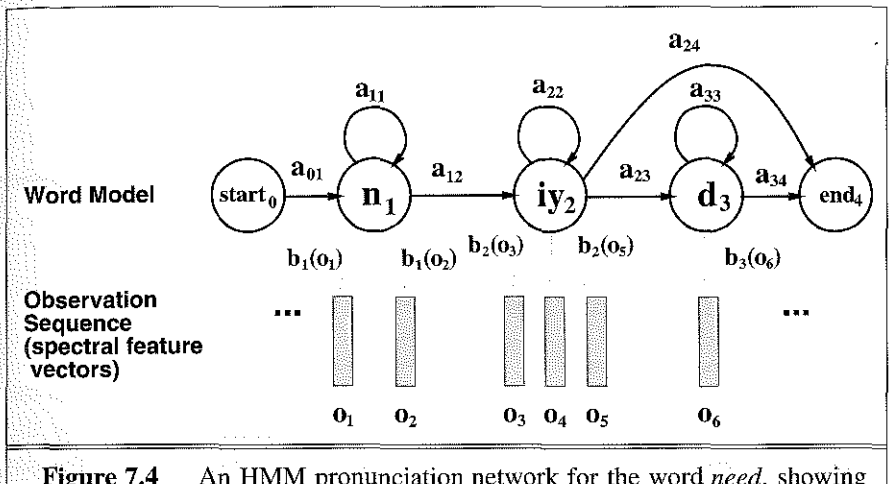


Figure 7.4 An HMM pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. Note the addition of the output probabilities B. HMMs used in speech recognition usually use self-loops on the states to model variable phone durations.

Figure 7.4 shows an HMM for the word *need* and a sample observation sequence. Note the differences from Figure 7.3. First, the observation sequences are now vectors of spectral features representing the speech signal. Next, note that we've also allowed one state to generate multiple copies of the same observation, by having a loop on the state. This loops allows HMMs to model the variable duration of phones; longer phones require more loops through the HMM.

In summary, here are the parameters we need to define an HMM:

- **states:** a set of states $Q = q_1 q_2 \ldots q_N$
- **transition probabilities:** a set of probabilities $A = a_{01} a_{02} \ldots a_{n1} \ldots a_{nn}$ Each a_{ij} represents the probability of transitioning from state i to state j. The set of these is the **transition probability matrix**
- **observation likelihoods:** a set of observation likelihoods $B = b_i(o_t)$, each expressing the probability of an observation o_t being generated from a state i

In our examples so far we have used two "special" states (**non-emitting states**) as the start and end state; as we saw in Chapter 5 it is also possible to avoid the use of these states by specifying two more things:

- **initial distribution:** an initial probability distribution over states, π, such that π_i is the probability that the HMM will start in state i. Of course some states j may have $\pi_j = 0$, meaning that they cannot be initial states.
- **accepting states:** a set of legal accepting states

As was true for the weighted automata, the sequences of symbols that are input to the model (if we are thinking of it as recognizer) or which are produced by the model (if we are thinking of it as a generator) are generally called the **observation sequence**, referred to as $O = (o_1 o_2 o_3 \ldots o_T)$.

7.3 The Viterbi Algorithm Revisited

Chapter 5 showed how the forward algorithm could be used to compute the probability of an observation sequence given an automaton, and how the Viterbi algorithm can be used to find the most-likely path through the automaton, as well as the probability of the observation sequence given this most-likely path. In Chapter 5 the observation sequences consisted of a single word. But in continuous speech, the input consists of sequences of words, and we are not given the location of the word boundaries. Knowing where the word boundaries are massively simplifies the problem of pronunciation; in Chapter 5, since we were sure that the pronunciation [ni] came from one word, we only had seven candidates to compare. But in actual speech we don't know where the word boundaries are. For example, try to decode the following sentence from Switchboard (don't peek ahead!):

[ay d ih s hh er d s ah m th ih ng ax b aw m uh v ih ng r ih s en l ih]

The answer is in the footnote.[2] The task is hard partly because of coarticulation and fast speech (e.g., [d] for the first phone of *just*!). But mainly it's the lack of spaces indicating word boundaries that make the task difficult. The task of finding word boundaries in connected speech is called **segmentation** and we will solve it by using the Viterbi algorithm just as we did for Chinese word-segmentation in Chapter 5; recall that the algorithm for Chinese word-segmentation relied on choosing the segmentation that resulted in the sequence of words with the highest frequency. For speech segmentation we use the more sophisticated N-gram language models introduced in Chapter 6. In the rest of this section we show how the Viterbi algorithm can

[2] I just heard something about moving recently.

be applied to the task of decoding and segmentation of a simple string of observations phones, using an n-gram language model. We will show how the algorithm is used to segment a very simple string of words. Here's the input and output we will work with:

Input	Output
[aa n iy dh ax]	*I need the*

Figure 7.5 shows word models for *I*, *need*, *the*, and also, just to make things difficult, the word *on*.

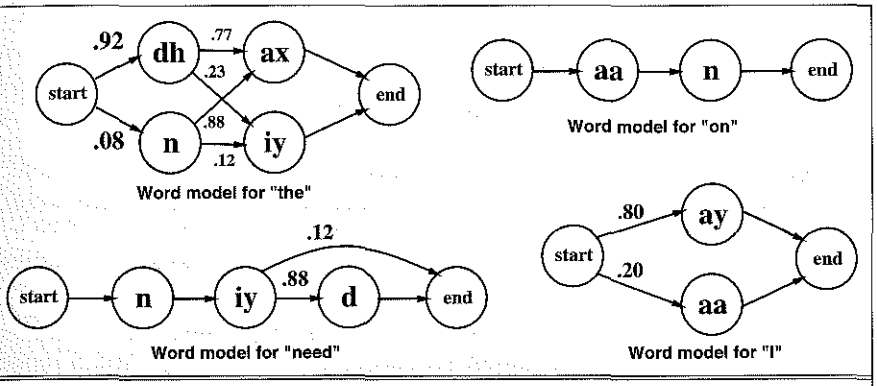


Figure 7.5 Pronunciation networks for the words *I*, *on*, *need*, and *the*. All networks (especially *the*) are significantly simplified.

Recall that the goal of the Viterbi algorithm is to find the best state sequence $q = (q_1 q_2 q_3 \ldots q_t)$ given the set of observed phones $o = (o_1 o_2 o_3 \ldots o_t)$. A graphic illustration of the output of the dynamic programming algorithm is shown in Figure 7.6. Along the y-axis are all the words in the lexicon; inside each word are its states. The x-axis is ordered by time, with one observed phone per time unit.[3] Each cell in the matrix will contain the probability of the most-likely sequence ending at that state. We can find the most-likely state sequence for the entire observation string by looking at the cell in the right-most column that has the highest probability, and tracing back the sequence that produced it.

[3] This x-axis component of the model is simplified in two major ways that we will show how to fix in the next section. First, the observations will not be phones but extracted spectral features, and second, each phone consists of not time unit observation but many observations (since phones can last for more than one phone). The y-axis is also simplified in this example, since as we will see most ASR system use multiple "subphone" units for each phone.

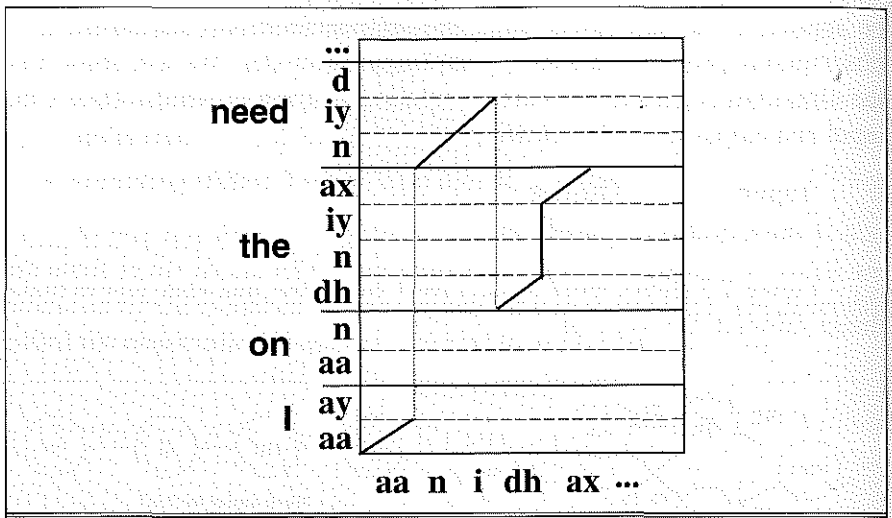


Figure 7.6 An illustration of the results of the Viterbi algorithm used to find the most-likely phone sequence (and hence estimate the most-likely word sequence).

More formally, we are searching for the best state sequence $q^* = (q_1 q_2 \dots q_T)$, given an observation sequence $o = (o_1 o_2 \dots o_T)$ and a model (a weighted automaton or "state graph") λ. Each cell *viterbi*[*i*,*t*] of the matrix contains the probability of the best path which accounts for the first t observations and ends in state i of the HMM. This is the most-probable path out of all possible sequences of states of length $t-1$:

$$viterbi[t,i] = \max_{q_1,q_2,\dots,q_{t-1}} P(q_1 q_2 \dots q_{t-1}, q_t = i, o_1, o_2 \dots o_t | \lambda) \qquad (7.8)$$

DYNAMIC PROGRAMMING INVARIANT

In order to compute *viterbi*[*t*,*i*], the Viterbi algorithm assumes the **dynamic programming invariant**. This is the simplifying (but incorrect) assumption that if the ultimate best path for the entire observation sequence happens to go through a state q_i, that this best path must include the best path up to and including state q_i. This doesn't mean that the best path at any time t is the best path for the whole sequence. A path can look bad at the beginning but turn out to be the best path. As we will see later, the Viterbi assumption breaks down for certain kinds of grammars (including trigram grammars) and so some recognizers have moved to another kind of decoder, the **stack** or **A*** decoder; more on that later. As we saw in our discussion of the minimum-edit-distance algorithm in Chapter 5, the reason for making the Viterbi assumption is that it allows us to break down the computation

of the optimal path probability in a simple way; each of the best paths at time t is the best extension of each of the paths ending at time $t-1$. In other words, the recurrence relation for the best path at time t ending in state j, $viterbi[t,j]$, is the maximum of the possible extensions of every possible previous path from time $t-1$ to time t:

$$viterbi[t,j] = \max_i (viterbi[t-1,i]\, a_{ij})\, b_j(o_t) \tag{7.9}$$

The algorithm as we describe it in Figure 7.9 takes a sequence of observations, and a single probabilistic automaton, and returns the optimal path through the automaton. Since the algorithm requires a single automaton, we will need to combine the different probabilistic phone networks for *the*, *I*, *need*, and *a* into one automaton. In order to build this new automaton we will need to add arcs with probabilities between any two words: bigram probabilities. Figure 7.7 shows simple bigram probabilities computed from the combined Brown and Switchboard corpus.

I need	0.0016	need need	0.000047	# Need	0.000018
I the	0.00018	need the	0.012	# The	0.016
I on	0.000047	need on	0.000047	# On	0.00077
I I	0.039	need I	0.000016	# I	0.079
the need	0.00051	on need	0.000055		
the the	0.0099	on the	0.094		
the on	0.00022	on on	0.0031		
the I	0.00051	on I	0.00085		

Figure 7.7 Bigram probabilities for the words *the*, *on*, *need*, and *I* following each other, and starting a sentence (i.e., following #). Computed from the combined Brown and Switchboard corpora with add-0.5 smoothing.

Figure 7.8 shows the combined pronunciation networks for the 4 words together with a few of the new arcs with the bigram probabilities. For readability of the diagram, most of the arcs aren't shown; the reader should imagine that each probability in Figure 7.7 is inserted as an arc between every two words.

The algorithm is given in Figure 5.19 in Chapter 5, and is repeated here for convenience as Figure 7.9. We see in Figure 7.9 that the Viterbi algorithm sets up a probability matrix, with one column for each time index t and one row for each state in the state graph. The algorithm first creates $T+2$ columns; Figure 7.9 shows the first six columns. The first column is an initial pseudo-observation, the next corresponds to the first observation

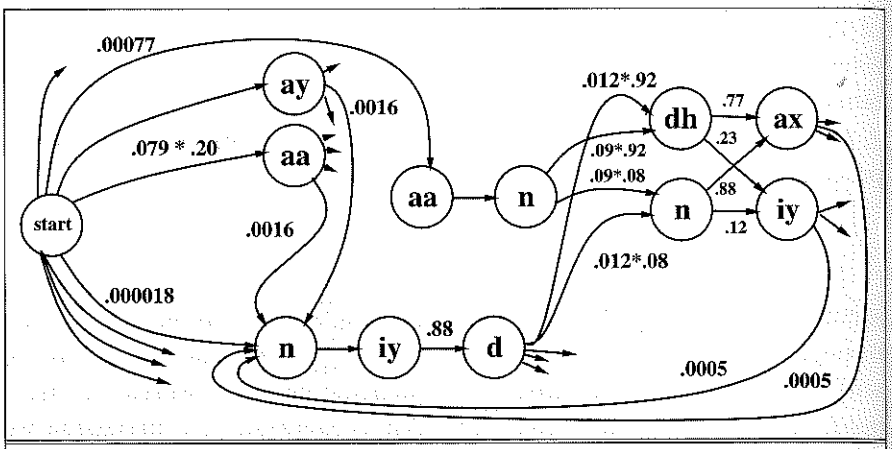


Figure 7.8 Single automaton made from the words *I*, *need*, *on*, and *the*. The arcs between words have probabilities computed from Figure 7.7. For lack of space the figure only shows a few of the between-word arcs.

phone [aa], and so on. We begin in the first column by setting the probability of the *start* state to 1.0, and the other probabilities to 0; the reader should find this in Figure 7.10. Cells with probability 0 are simply left blank for readability. For each column of the matrix, that is, for each time index t, each cell $viterbi[t,j]$, will contain the probability of the most likely path to end in that cell. We will calculate this probability recursively, by maximizing over the probability of coming from all possible preceding states. Then we move to the next state; for each of the i states $viterbi[0,i]$ in column 0, we compute the probability of moving into each of the j states $viterbi[1,j]$ in column 1, according to the recurrence relation in (7.9). In the column for the input *aa*, only two cells have non-zero entries, since $b_1(aa)$ is zero for every other state except the two states labeled *aa*. The value of $viterbi(1,aa)$ of the word *I* is the product of the transition probability from # to *I* and the probability of *I* being pronounced with the vowel *aa*.

Notice that if we look at the column for the observation *n*, that the word *on* is currently the "most-probable" word. But since there is no word or set of words in this lexicon which is pronounced *i dh ax*, the path starting with *on* is a dead end, that is, this hypothesis can never be extended to cover the whole utterance.

By the time we see the observation *iy*, there are two competing paths: *I need* and *I the*; *I need* is currently more likely. When we get to the observation *dh*, we could have arrived from either the *iy* of *need* or the *iy* of *the*.

```
function VITERBI(observations of len T,state-graph) returns best-path

  num-states ← NUM-OF-STATES(state-graph)
  Create a path probability matrix viterbi[num-states+2,T+2]
  viterbi[0,0] ← 1.0
  for each time step t from 0 to T do
    for each state s from 0 to num-states do
      for each transition s' from s specified by state-graph
        new-score ← viterbi[s, t] * a[s,s'] * b_s'(o_t)
        if ((viterbi[s',t+1] = 0) || (new-score > viterbi[s', t+1]))
          then
                viterbi[s', t+1] ← new-score
                back-pointer[s', t+1] ← s
  Backtrace from highest probability state in the final column of viterbi[] and
return path.
```

Figure 7.9 Viterbi algorithm for finding optimal sequence of states in continuous speech recognition, simplified by using phones as inputs (duplicate of Figure 5.19). Given an observation sequence of phones and a weighted automaton (state graph), the algorithm returns the path through the automaton which has minimum probability and accepts the observation sequence. $a[s,s']$ is the transition probability from current state s to next state s' and $b_{s'}(o_t)$ is the observation likelihood of s' given o_t.

The probability of the *max* of these two paths, in this case the path through *I need*, will go into the cell for *dh*.

Finally, the probability for the best path will appear in the final *ax* column. In this example, only one cell is non-zero in this column; the *ax* state of the word *the* (a real example wouldn't be this simple; many other cells would be non-zero).

If the sentence had actually ended here, we would now need to backtrace to find the path that gave us this probability. We can't just pick the highest probability state for each state column. Why not? Because the most likely path early on is not necessarily the most likely path for the whole sentence. Recall that the most likely path after seeing *n* was the word *on*. But the most likely path for the whole sentence is *I need the*. Thus we had to rely in Figure 7.10 on the "Hansel and Gretel" method (or the "Jason and the Minotaur" method if you like your metaphors more classical): whenever we moved into a cell, we kept pointers back to the cell we came from. The reader should convince themselves that the Viterbi algorithm has simultaneously solved the segmentation and decoding problems.

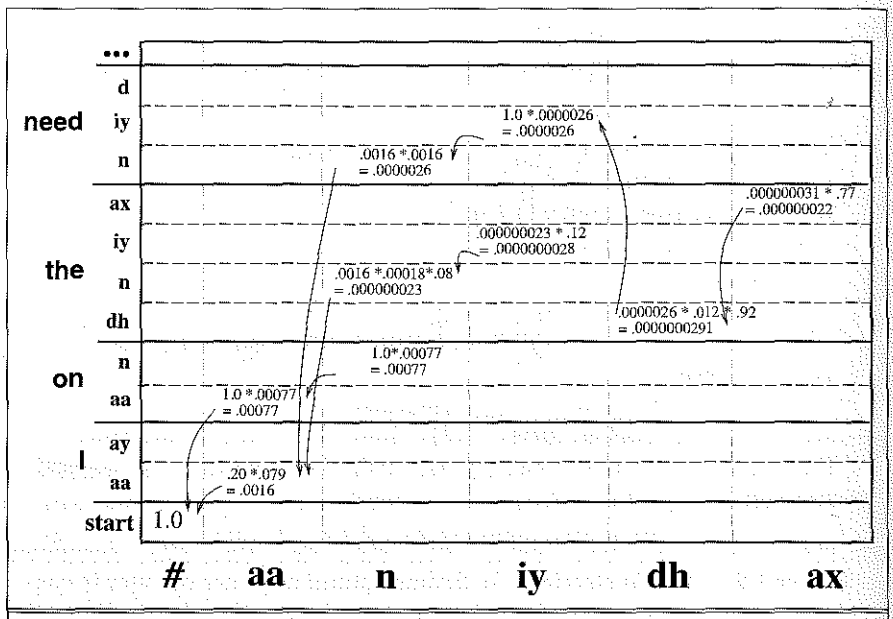


Figure 7.10 The entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. Backtracing from the successful last word (*the*), we can reconstruct the word sequence *I need the*.

The presentation of the Viterbi algorithm in this section has been simplified; actual implementations of Viterbi decoding are more complex in three key ways that we have mentioned already. First, in an actual HMM for speech recognition, the input would not be phones. Instead, the input is a **feature vector** of spectral and acoustic features. Thus the **observation likelihood probabilities** $b_i(t)$ of an observation o_t given a state i will not simply take on the values 0 or 1, but will be more fine-grained probability estimates, computed via mixtures of Gaussian probability estimators or neural nets. The next section will show how these probabilities are computed.

Second, the HMM states in most speech recognition systems are not simple phones but rather **subphones**. In these systems each phone is divided into three states: the beginning, middle and final portions of the phone. Dividing up a phone in this way captures the intuition that the significant changes in the acoustic input happen at a finer granularity than the phone; for example the closure and release of a stop consonant. Furthermore, many systems use a separate instance of each of these subphones for each **triphone** context (Schwartz et al., 1985; Deng et al., 1990). Thus instead of around

TRIPHONE

60 phone units, there could be as many as 60^3 context-dependent triphones. In practice, many possible sequences of phones never occur or are very rare, so systems create a much smaller number of triphones models by **clustering** the possible triphones (Young and Woodland, 1994). Figure 7.11 shows an example of the complete phone model for the triphone b(ax,aw).

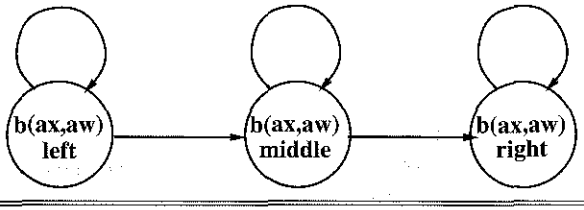


Figure 7.11 An example of the context-dependent triphone b(ax,aw) (the phone [b] preceded by a [ax] and followed by a [aw], as in the beginning of *about*, showing its left, middle, and right subphones.

Finally, in practice in large-vocabulary recognition it is too expensive to consider all possible words when the algorithm is extending paths from one state-column to the next. Instead, low-probability paths are pruned at each time step and not extended to the next state column. This is usually implemented via **beam search**: for each state column (time step), the algorithm maintains a short list of high-probability words whose path probabilities are within some percentage (**beam width**) of the most probable word path. Only transitions from these words are extended when moving to the next time step. Since the words are ranked by the probability of the path so far, which words are within the beam (active) will change from time step to time step. Making this beam search approximation allows a significant speed-up at the cost of a degradation to the decoding performance. This beam search strategy was first implemented by Lowerre (1968). Because in practice most implementations of Viterbi use beam search, some of the literature uses the term **beam search** or **time-synchronous beam search** instead of Viterbi.

BEAM SEARCH

BEAM WIDTH

7.4 Advanced Methods for Decoding

There are two main limitations of the Viterbi decoder. First, the Viterbi decoder does not actually compute the sequence of words which is most probable given the input acoustics. Instead, it computes an approximation to this: the sequence of *states* (i.e., *phones* or *subphones*) which is most prob-

able given the input. This difference may not always be important; the most probable sequence of phones may very well correspond exactly to the most probable sequence of words. But sometimes the most probable sequence of phones does not correspond to the most probable word sequence. For example consider a speech recognition system whose lexicon has multiple pronunciations for each word. Suppose the correct word sequence includes a word with very many pronunciations. Since the probabilities leaving the start arc of each word must sum to 1.0, each of these pronunciation-paths through this multiple-pronunciation HMM word model will have a smaller probability than the path through a word with only a single pronunciation path. Thus because the Viterbi decoder can only follow one of these pronunciation paths, it may ignore this word in favor of an incorrect word with only one pronunciation path.

A second problem with the Viterbi decoder is that it cannot be used with all possible language models. In fact, the Viterbi algorithm as we have defined it cannot take complete advantage of any language model more complex than a bigram grammar. This is because of the fact mentioned early that a trigram grammar, for example, violates the **dynamic programming invariant** that makes dynamic programming algorithms possible. Recall that this invariant is the simplifying (but incorrect) assumption that if the ultimate best path for the entire observation sequence happens to go through a state q_i, that this best path must include the best path up to and including state q_i. Since a trigram grammar allows the probability of a word to be based on the two previous words, it is possible that the best trigram-probability path for the sentence may go through a word but not include the best path to that word. Such a situation could occur if a particular word w_x has a high trigram probability given w_y, w_z, but that conversely the best path to w_y didn't include w_z (i.e., $P(w_y|w_q, w_z)$ was low for all q).

There are two classes of solutions to these problems with Viterbi decoding. One class involves modifying the Viterbi decoder to return multiple potential utterances and then using other high-level language model or pronunciation-modeling algorithms to re-rank these multiple outputs. In general this kind of **multiple-pass decoding** allows a computationally efficient, but perhaps unsophisticated, language model like a bigram to perform a rough first decoding pass, allowing more sophisticated but slower decoding algorithms to run on a reduced search space.

N-BEST

For example, Schwartz and Chow (1990) give a Viterbi-like algorithm which returns the **N-best** sentences (word sequences) for a given speech input. Suppose for example a bigram grammar is used with this N-best-Viterbi

to return the 10,000 most highly-probable sentences, each with their likelihood score. A trigram-grammar can then be used to assign a new language-model prior probability to each of these sentences. These priors can be combined with the acoustic likelihood of each sentence to generate a posterior probability for each sentence. Sentences can then be **rescored** using this more sophisticated probability. Figure 7.12 shows an intuition for this algorithm. RESCORED

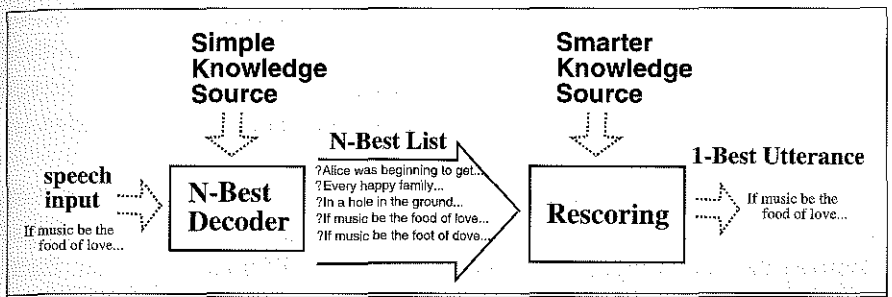


Figure 7.12 The use of N-best decoding as part of a two-stage decoding model. Efficient but unsophisticated knowledge sources are used to return the N-best utterances. This significantly reduces the search space for the second pass models, which are thus free to be very sophisticated but slow.

An augmentation of N-best, still part of this first class of extensions to Viterbi, is to return, not a list of sentences, but a **word lattice**. A word lattice is a directed graph of words and links between them which can compactly encode a large number of possible sentences. Each word in the lattice is augmented with its observation likelihood, so that any particular path through the lattice can then be combined with the prior probability derived from a more sophisticated language model. For example Murveit et al. (1993) describe an algorithm used in the SRI recognizer Decipher which uses a bigram grammar in a rough first pass, producing a word lattice which is then refined by a more sophisticated language model. WORD LATTICE

The second solution to the problems with Viterbi decoding is to employ a completely different decoding algorithm. The most common alternative algorithm is the **stack decoder**, also called the **A*** decoder (Jelinek, 1969; Jelinek et al., 1975). We will describe the algorithm in terms of the **A*** **search** used in the artificial intelligence literature, although the development of stack decoding actually came from the communications theory literature and the link with AI best-first search was noticed only later (Jelinek, 1976). STACK DECODER A* A* SEARCH

A* Decoding

To see how the A* decoding method works, we need to revisit the Viterbi algorithm. Recall that the Viterbi algorithm computed an approximation of the forward algorithm. Viterbi computes the observation likelihood of the single best (MAX) path through the HMM, while the forward algorithm computes the observation likelihood of the total (SUM) of all the paths through the HMM. But we accepted this approximation because Viterbi computed this likelihood *and* searched for the optimal path simultaneously. The A* decoding algorithm, on the other hand, will rely on the complete forward algorithm rather than an approximation. This will ensure that we compute the correct observation likelihood. Furthermore, the A* decoding algorithm allows us to use any arbitrary language model.

The A* decoding algorithm is a kind of best-first search of the lattice or tree which implicitly defines the sequence of allowable words in a language. Consider the tree in Figure 7.13, rooted in the START node on the left. Each leaf of this tree defines one sentence of the language; the one formed by concatenating all the words along the path from START to the leaf. We don't represent this tree explicitly, but the stack decoding algorithm uses the tree implicitly as a way to structure the decoding search.

The algorithm performs a search from the root of the tree toward the leaves, looking for the highest probability path, and hence the highest probability sentence. As we proceed from root toward the leaves, each branch leaving a given word node represent a word which may follow the current word. Each of these branches has a probability, which expresses the conditional probability of this next word given the part of the sentence we've seen so far. In addition, we will use the forward algorithm to assign each word a likelihood of producing some part of the observed acoustic data. The A* decoder must thus find the path (word sequence) from the root to a leaf which has the highest probability, where a path probability is defined as the product of its language model probability (prior) and its acoustic match to the data (likelihood). It does this by keeping a **priority queue** of partial paths (i.e., prefixes of sentences, each annotated with a score). In a priority queue each element has a score, and the *pop* operation returns the element with the highest score. The A* decoding algorithm iteratively chooses the best prefix-so-far, computes all the possible next words for that prefix, and adds these extended sentences to the queue. The Figure 7.14 shows the complete algorithm.

PRIORITY QUEUE

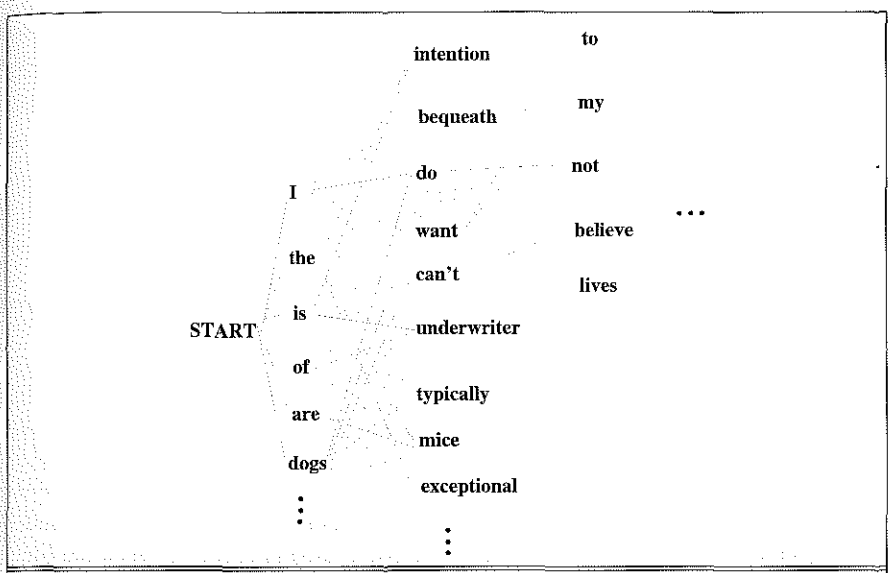


Figure 7.13 A visual representation of the implicit lattice of allowable word sequences that defines a language. The set of sentences of a language is far too large to represent explicitly, but the lattice gives a metaphor for exploring substrings of these sentences.

Let's consider a stylized example of a A* decoder working on a waveform for which the correct transcription is *If music be the food of love*. Figure 7.15 shows the search space after the decoder has examined paths of length one from the root. A **fast match** is used to select the likely next words. A fast match is one of a class of heuristics designed to efficiently winnow down the number of possible following words, often by computing some approximation to the forward probability (see below for further discussion of fast matching). FAST MATCH

At this point in our example, we've done the fast match, selected a subset of the possible next words, and assigned each of them a score. The word *Alice* has the highest score. We haven't yet said exactly how the scoring works, although it will involve as a component the probability of the hypothesized sentence given the acoustic input $P(W|A)$, which itself is composed of the language model probability $P(W)$ and the acoustic likelihood $P(A|W)$.

Figure 7.16 show the next stage in the search. We have expanded the *Alice* node. This means that the *Alice* node is no longer on the queue, but its children are. Note that now the node labeled *if* actually has a higher score than any of the children of *Alice*.

function STACK-DECODING() **returns** *min-distance*

Initialize the priority queue with a null sentence.
Pop the best (highest score) sentence s off the queue.
If (s is marked end-of-sentence (EOS)) output s and terminate.
Get list of candidate next words by doing fast matches.
For each candidate next word w:
    Create a new candidate sentence $s + w$.
    Use forward algorithm to compute acoustic likelihood L of $s + w$
    Compute language model probability P of extended sentence $s + w$
    Compute "score" for $s + w$ (a function of L, P, and ???)
    if (end-of-sentence) set EOS flag for $s + w$.
    Insert $s + w$ into the queue together with its score and EOS flag

Figure 7.14 The A* decoding algorithm (modified from Paul (1991) and Jelinek (1997)). The evaluation function that is used to compute the score for a sentence is not completely defined here; possibly evaluation functions are discussed below.

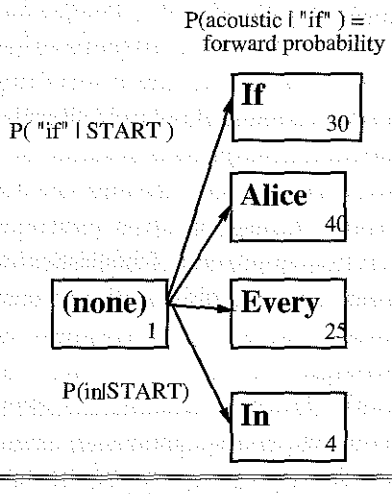


Figure 7.15 The beginning of the search for the sentence *If music be the food of love*. At this early stage *Alice* is the most likely hypothesis. (It has a higher score than the other hypotheses.)

Figure 7.17 shows the state of the search after expanding the *if* node, removing it, and adding *if music*, *if muscle*, and *if messy* on to the queue.

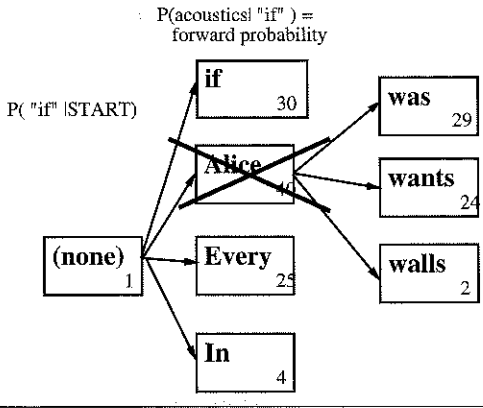


Figure 7.16 The next step of the search for the sentence *If music be the food of love*. We've now expanded the *Alice* node and added three extensions which have a relatively high score (*was*, *wants*, and *walls*). Note that now the node with the highest score is *START if*, which is not along the *START Alice* path at all!

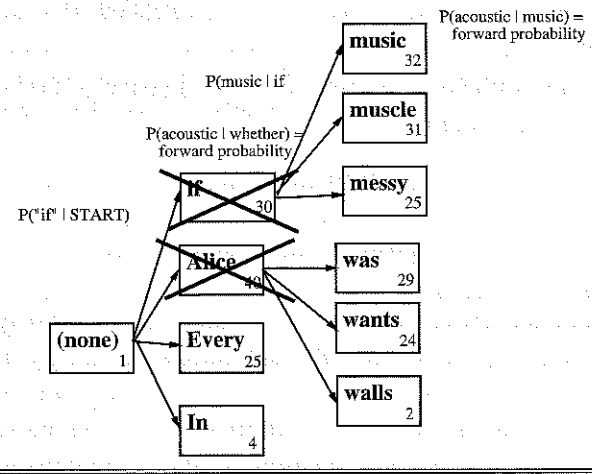


Figure 7.17 We've now expanded the *if* node. The hypothesis *START if music* currently has the highest score.

We've implied that the scoring criterion for a hypothesis is related to its probability. Indeed it might seem that the score for a string of words w_1^i given an acoustic string y_1^j should be the product of the prior and the likelihood:

$$P(y_1^j|w_1^i)P(w_1^i)$$

Alas, the score cannot be this probability because the probability will be much smaller for a longer path than a shorter one. This is due to a simple fact about probabilities and substrings; any prefix of a string must have a higher probability than the string itself (e.g., P(START the ...) will be greater than P(START the book)). Thus if we used probability as the score, the A* decoding algorithm would get stuck on the single-word hypotheses.

Instead, we use what is called the A* evaluation function (Nilsson, 1980; Pearl, 1984) called $f^*(p)$, given a partial path p:

$$f^*(p) = g(p) + h^*(p)$$

$f^*(p)$ is the *estimated* score of the best complete path (complete sentence) which starts with the partial path p. In other words, it is an estimate of how well this path would do if we let it continue through the sentence. The A* algorithm builds this estimate from two components:

- $g(p)$ is the score from the beginning of utterance to the end of the partial path p. This g function can be nicely estimated by the probability of p given the acoustics so far (i.e., as $P(A|W)P(W)$ for the word string W constituting p).
- $h^*(p)$ is an estimate of the best scoring extension of the partial path to the end of the utterance.

Coming up with a good estimate of h^* is an unsolved and interesting problem. One approach is to choose as h^* an estimate which correlates with the number of words remaining in the sentence (Paul, 1991); see Jelinek (1997) for further discussion.

We mentioned above that both the A* and various other two-stage decoding algorithms require the use of a **fast match** for quickly finding which words in the lexicon are likely candidates for matching some portion of the acoustic input. Many fast match algorithms are based on the use of a **tree-structured lexicon**, which stores the pronunciations of all the words in such a way that the computation of the forward probability can be shared for words which start with the same sequence of phones. The tree-structured lexicon was first suggested by Klovstad and Mondshein (1975); fast match algorithms which make use of it include Gupta et al. (1988), Bahl et al. (1992) in the context of A* decoding, and Ney et al. (1992) and Nguyen and Schwartz (1999) in the context of Viterbi decoding. Figure 7.18 shows an example of a tree-structured lexicon from the Sphinx-II recognizer (Ravishankar, 1996). Each tree root represents the first phone of all words begin-

TREE-STRUCTURED LEXICON

ning with that context dependent phone (phone context may or may not be preserved across word boundaries), and each leaf is associated with a word.

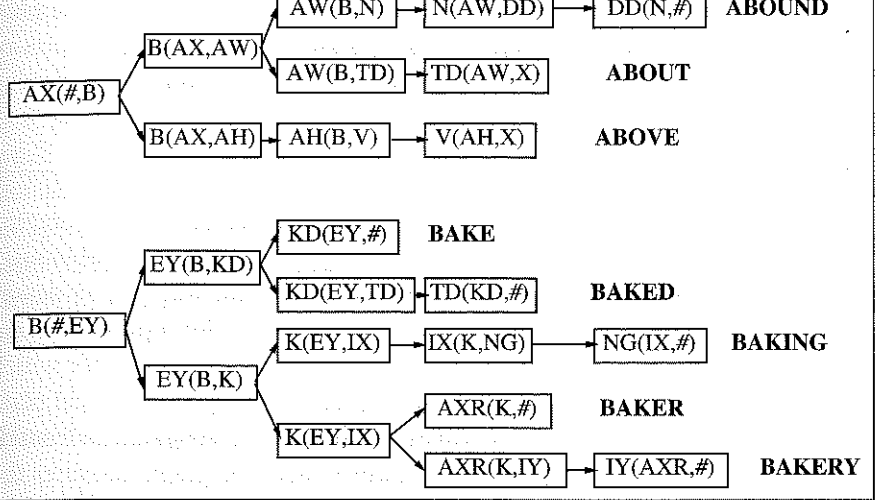


Figure 7.18 A tree-structured lexicon from the Sphinx-II recognizer (after Ravishankar (1996)). Each node corresponds to a particular triphone in a slightly modified version of the ARPAbet; thus EY(B,KD) means the phone EY preceded by a B and followed by the closure of a K.

There are many other kinds of multiple-stage search, such as the **forward-backward** search algorithm (not to be confused with the **forward-backward** algorithm for HMM parameter setting) (Austin et al., 1991) which performs a simple forward search followed by a detailed backward (i.e., time-reversed) search.

FORWARD-BACKWARD

7.5 Acoustic Processing of Speech

This section presents a very brief overview of the kind of acoustic processing commonly called **feature extraction** or **signal analysis** in the speech recognition literature. The term **features** refers to the vector of numbers which represent one time-slice of a speech signal. A number of kinds of features are commonly used, such as **LPC** features and **PLP** features. All of these are **spectral features**, which means that they represent the waveform in terms of the distribution of different **frequencies** which make up the waveform; such a distribution of frequencies is called a **spectrum**. We will begin with a brief

FEATURE EXTRACTION
SIGNAL ANALYSIS
LPC
PLP
SPECTRAL FEATURES

introduction to the acoustic waveform and how it is digitized, summarize the idea of frequency analysis and spectra, and then sketch out different kinds of extracted features. This will be an extremely brief overview; the interested reader should refer to other books on the linguistics aspects of acoustic phonetics (Johnson, 1997; Ladefoged, 1996) or on the engineering aspects of digital signal processing of speech (Rabiner and Juang, 1993).

Sound Waves

The input to a speech recognizer, like the input to the human ear, is a complex series of changes in air pressure. These changes in air pressure obviously originate with the speaker, and are caused by the specific way that air passes through the glottis and out the oral or nasal cavities. We represent sound waves by plotting the change in air pressure over time. One metaphor which sometimes helps in understanding these graphs is to imagine a vertical plate which is blocking the air pressure waves (perhaps in a microphone in front of a speaker's mouth, or the eardrum in a hearer's ear). The graph measures the amount of **compression** or **rarefaction** (uncompression) of the air molecules at this plate. Figure 7.19 shows the waveform taken from the Switchboard corpus of telephone speech of someone saying "she just had a baby".

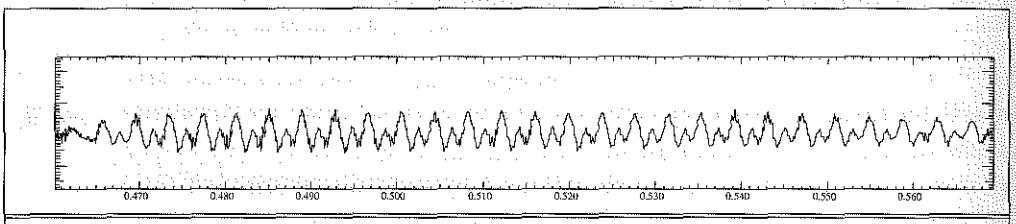


Figure 7.19 A waveform of the vowel [iy] from the utterance shown in Figure 7.20. The *y*-axis shows the changes in air pressure above and below normal atmospheric pressure. The *x*-axis shows time. Notice that the wave repeats regularly.

FREQUENCY
AMPLITUDE

Two important characteristics of a wave are its **frequency** and **amplitude**. The frequency is the number of times a second that a wave repeats itself, or **cycles**. Note in Figure 7.19 that there are 28 repetitions of the wave in the .11 seconds we have captured. Thus the frequency of this segment of the wave is 28/.11 or 255 **cycles per second**. Cycles per second are usually called **Hertz** (shortened to **Hz**), so the frequency in Figure 7.19 would be described as 255 Hz.

CYCLES PER SECOND
HERTZ

The vertical axis in Figure 7.19 measures the amount of air pressure

variation. A high value on the vertical axis (a high **amplitude**) indicates that there is more air pressure at that point in time, a zero value means there is normal (atmospheric) air pressure, while a negative value means there is lower than normal air pressure (rarefaction). AMPLITUDE

Two important perceptual properties are related to frequency and amplitude. The **pitch** of a sound is the perceptual correlate of frequency; in general if a sound has a higher frequency we perceive it as having a higher pitch, although the relationship is not linear, since human hearing has different acuities for different frequencies. Similarly, the **loudness** of a sound is the perceptual correlate of the **power**, which is related to the square of the amplitude. So sounds with higher amplitudes are perceived as louder, but again the relationship is not linear. PITCH

How to Interpret a Waveform

Since humans (and to some extent machines) can transcribe and understand speech just given the sound wave, the waveform must contain enough information to make the task possible. In most cases this information is hard to unlock just by looking at the waveform, but such visual inspection is still sufficient to learn some things. For example, the difference between vowels and most consonants is relatively clear on a waveform. Recall that vowels are voiced, tend to be long, and are relatively loud. Length in time manifests itself directly as length in space on a waveform plot. Loudness manifests itself as high amplitude. How do we recognize voicing? Recall that voicing is caused by regular openings and closing of the vocal folds. When the vocal folds are vibrating, we can see regular peaks in amplitude of the kind we saw in Figure 7.19. During a stop consonant, for example the closure of a [p], [t], or [k], we should expect no peaks at all; in fact we expect silence.

Notice in Figure 7.20 the places where there are regular amplitude peaks indicating voicing; from second .46 to .58 (the vowel [iy]), from second .65 to .74 (the vowel [ax]) and so on. The places where there is no amplitude indicate the silence of a stop closure; for example from second 1.06 to second 1.08 (the closure for the first [b], or from second 1.26 to 1.28 (the closure for the second [b]).

Fricatives like [sh] can also be recognized in a waveform; they produce an intense irregular pattern; the [sh] from second .33 to .46 is a good example of a fricative.

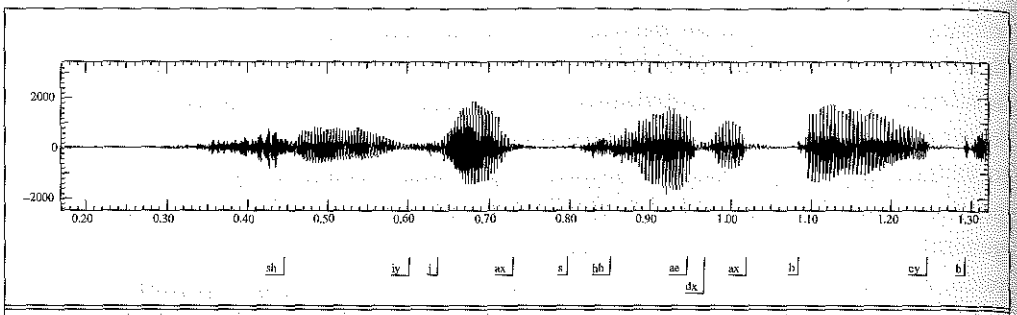


Figure 7.20 A waveform of the sentence "She just had a baby" from the Switchboard corpus (conversation 4325). The speaker is female, was 20 years old in 1991, which is approximately when the recording was made, and speaks the South Midlands dialect of American English. The phone labels show where each phone ends. The last bit of the final [iy] vowel is cut off in this figure.

Spectra

While some broad phonetic features (presence of voicing, stop closures, fricatives) can be interpreted from a waveform, more detailed classification (which vowel? which fricative?) requires a different representation of the input in terms of **spectral** features. Spectral features are based on the insight of Fourier that every complex wave can be represented as a sum of many simple waves of different frequencies. A musical analogy for this is the chord; just as a chord is composed of multiple notes, any waveform is composed of the waves corresponding to its individual "notes".

SPECTRAL

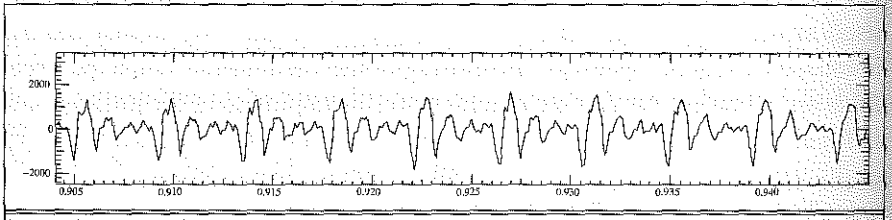


Figure 7.21 The waveform of part of the vowel [æ] from the word *had* cut out from the waveform shown in Figure 7.20.

Consider Figure 7.21, which shows part of the waveform for the vowel [æ] of the word *had* at second 0.9 of the sentence. Note that there is a complex wave which repeats about nine times in the figure; but there is also a smaller repeated wave which repeats four times for every larger pattern (notice the four small peaks inside each repeated wave). The complex wave has

a frequency of about 250 Hz (we can figure this out since it repeats roughly 9 times in .036 seconds, and 9 cycles/.036 seconds = 250 Hz). The smaller wave then should have a frequency of roughly four times the frequency of the larger wave, or roughly 1000 Hz. Then if you look carefully you can see two little waves on the peak of many of the 1000 Hz waves. The frequency of this tiniest wave must be roughly twice that of the 1000 Hz wave, hence 2000 Hz.

SPECTRUM
FOURIER TRANSFORM

A **spectrum** is a representation of these different frequency components of a wave. It can be computed by a **Fourier transform**, a mathematical procedure which separates out each of the frequency components of a wave. Rather than using the Fourier transform spectrum directly, most speech applications use a smoothed version of the spectrum called the **LPC** spectrum (Atal and Hanauer, 1971; Itakura, 1975).

LPC

Figure 7.22 shows an LPC spectrum for the waveform in Figure 7.21. LPC (**Linear Predictive Coding**) is a way of coding the spectrum that makes it easier to see where the **spectral peaks** are.

SPECTRAL PEAKS

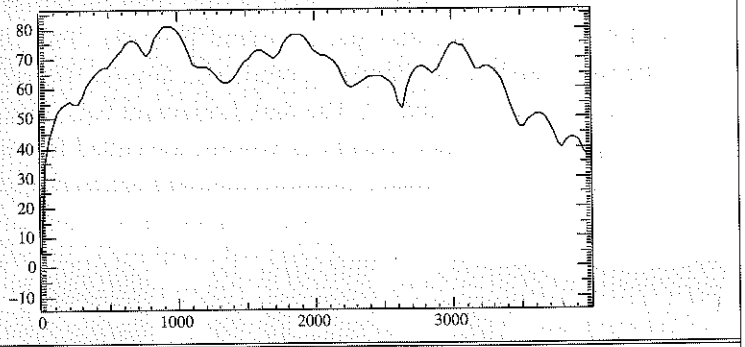


Figure 7.22 An LPC spectrum for the vowel [æ] waveform of *She just had a baby* at the point in time shown in Figure 7.21. LPC makes it easy to see **formants**.

The x-axis of a spectrum shows frequency while the y-axis shows some measure of the magnitude of each frequency component (in decibels (dB), a logarithmic measure of amplitude). Thus Figure 7.22 shows that there are important frequency components at 930 Hz, 1860 Hz, and 3020 Hz, along with many other lower-magnitude frequency components. These important components at roughly 1000 Hz and 2000 Hz are just what we predicted by looking at the wave in Figure 7.21!

Why is a spectrum useful? It turns out that these spectral peaks that are easily visible in a spectrum are very characteristic of different sounds; phones have characteristic spectral "signatures". For example different chemical elements give off different wavelengths of light when they burn, allowing us to detect elements in stars light-years away by looking at the spectrum of the light. Similarly, by looking at the spectrum of a waveform, we can detect the characteristic signature of the different phones that are present. This use of spectral information is essential to both human and machine speech recognition. In human audition, the function of the **cochlea** or **inner ear** is to compute a spectrum of the incoming waveform. Similarly, the features used as input to the HMMs in speech recognition are all representations of spectra, usually variants of LPC spectra, as we will see.

COCHLEA

INNER EAR

While a spectrum shows the frequency components of a wave at one point in time, a **spectrogram** is a way of envisioning how the different frequencies which make up a waveform change over time. The x-axis shows time, as it did for the waveform, but the y-axis now shows frequencies in Hertz. The darkness of a point on a spectrogram corresponding to the amplitude of the frequency component. For example, look in Figure 7.23 around second 0.9 and notice the dark bar at around 1000 Hz. This means that the [iy] of the word *she* has an important component around 1000 Hz (1000 Hz is just between the notes B and C). The dark horizontal bars on a spectrogram, representing spectral peaks, usually of vowels, are called **formants**.

SPECTROGRAM

FORMANTS

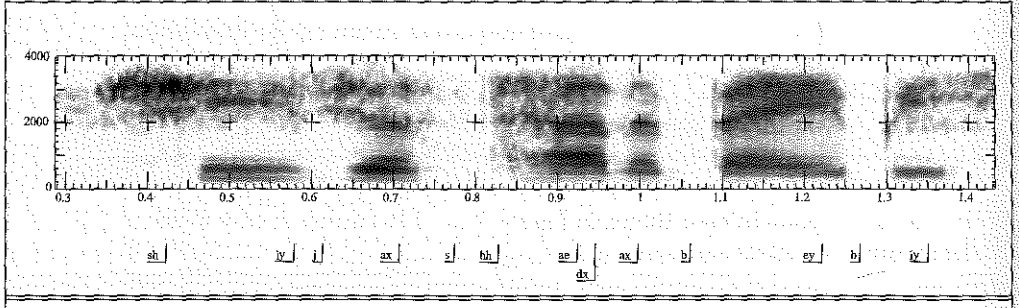


Figure 7.23 A spectrogram of the sentence "She just had a baby" whose waveform was shown in Figure 7.20. One way to think of a spectrogram is as a collection of spectra (time-slices) like Figure 7.22 placed end to end.

What specific clues can spectral representations give for phone identification? First, different vowels have their formants at characteristic places. We've seen that [æ] in the sample waveform had formants at 930 Hz, 1860 Hz, and 3020 Hz. Consider the vowel [iy], at the beginning of the utterance

in Figure 7.20. The spectrum for this vowel is shown in Figure 7.24. The first formant of [iy] is 540 Hz; much lower than the first formant for [æ], while the second formant (2581 Hz) is much higher than the second formant for [æ]. If you look carefully you can see these formants as dark bars in Figure 7.23 just around 0.5 seconds.

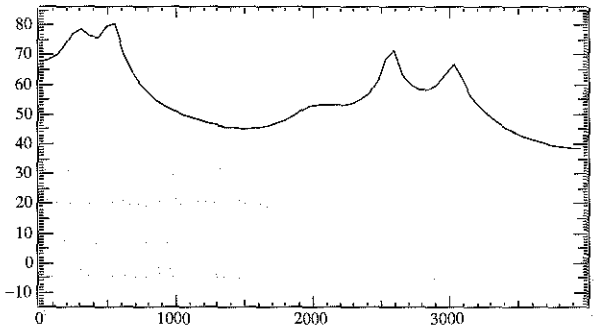


Figure 7.24 A smoothed (LPC) spectrum for the vowel [iy] at the start of *She just had a baby*. Note that the first formant (540 Hz) is much lower than the first formant for [æ] shown in Figure 7.22, while the second formant (2581 Hz) is much higher than the second formant for [æ].

The location of the first two formants (called F1 and F2) plays a large role in determining vowel identity, although the formants still differ from speaker to speaker. Formants also can be used to identify the nasal phones [n], [m], and [ŋ], the lateral phone [l], and [r]. Why do different vowels have different spectral signatures? The formants are caused by the resonant cavities of the mouth. The oral cavity can be thought of as a filter which selectively passes through some of the harmonics of the vocal cord vibrations. Moving the tongue creates spaces of different size inside the mouth which selectively amplify waves of the appropriate wavelength, hence amplifying different frequency bands.

Feature Extraction

Our survey of the features of waveforms and spectra was necessarily brief, but the reader should have the basic idea of the importance of spectral features and their relation to the original waveform. Let's now summarize the process of extraction of spectral features, beginning with the sound wave

itself and ending with a **feature vector**.[4] An input soundwave is first **digitized**. This process of **analog-to-digital conversion** has two steps: **sampling** and **quantization**. A signal is sampled by measuring its amplitude at a particular time; the **sampling rate** is the number of samples taken per second. Common sampling rates are 8,000 Hz and 16,000 Hz. In order to accurately measure a wave, it is necessary to have at least two samples in each cycle: one measuring the positive part of the wave and one measuring the negative part. More than two samples per cycle increases the amplitude accuracy, but less than two samples will cause the frequency of the wave to be completely missed. Thus the maximum frequency wave that can be measured is one whose frequency is half the sample rate (since every cycle needs two samples). This maximum frequency for a given sampling rate is called the **Nyquist frequency**. Most information in human speech is in frequencies below 10,000 Hz; thus a 20,000 Hz sampling rate would be necessary for complete accuracy. But telephone speech is filtered by the switching network, and only frequencies less than 4,000 Hz are transmitted by telephones. Thus an 8,000 Hz sampling rate is sufficient for telephone-bandwidth speech like the Switchboard corpus.

SAMPLING

SAMPLING RATE

NYQUIST FREQUENCY

Even an 8,000 Hz sampling rate requires 8000 amplitude measurements for each second of speech, and so it is important to store the amplitude measurement efficiently. They are usually stored as integers, either 8-bit (values from -128–127) or 16 bit (values from -32768–32767). This process of representing a real-valued number as a integer is called **quantization** because there is a minimum granularity (the quantum size) and all values which are closer together than this quantum size are represented identically.

QUANTIZATION

Once a waveform has been digitized, it is converted to some set of spectral features. An LPC spectrum is represented by a vector of features; each formant is represented by two features, plus two additional features to represent spectral tilt. Thus five formants can be represented by 12 ($5 \times 2 + 2$) features. It is possible to use LPC features directly as the observation symbols of an HMM. However, further processing is often done to the features. One popular feature set is **cepstral**, which are computed from the LPC coefficients by taking the Fourier transform of the spectrum. Another feature set, **PLP (Perceptual Linear Predictive** analysis (Hermansky, 1990)), takes the LPC features and modifies them in ways consistent with human hearing. For

CEPSTRAL COEFFICIENTS

PLP

[4] The reader might want to bear in mind Picone's (1993) reminder that the use of the word **extraction** should not be thought of as encouraging the metaphor of features as something "in the signal" waiting to be extracted.

example, the spectral resolution of human hearing is worse at high frequencies, and the perceived loudness of a sound is related to the cube rate of its intensity. So PLP applies various filters to the LPC spectrum and takes the cube root of the features.

7.6 Computing Acoustic Probabilities

The last section showed how the speech input can be passed through signal processing transformations and turned into a series of vectors of features, each vector representing one time-slice of the input signal. How are these feature vectors turned into probabilities?

CLUSTER

One way to compute probabilities on feature vectors is to first **cluster** them into discrete symbols that we can count; we can then compute the probability of a given cluster just by counting the number of times it occurs in some training set. This method is usually called **vector quantization**. Vector quantization was quite common in early speech recognition algorithms but has mainly been replaced by a more direct but compute-intensive approach: computing observation probabilities on a real-valued ('continuous') input vector. This method thus computes a **probability density function** or **pdf** over a continuous space.

VECTOR QUANTIZATION

PROBABILITY DENSITY FUNCTION

There are two popular versions of the continuous approach. The most widespread of the two is the use of **Gaussian** pdfs, in the simplest version of which each state has a single Gaussian function which maps the observation vector o_t to a probability. An alternative approach is the use of **neural networks** or **multi-layer perceptrons** which can also be trained to assign a probability to a real-valued feature vector. HMMs with Gaussian observation-probability-estimators are trained by a simple extension to the forward-backward algorithm (discussed in Appendix D). HMMs with neural-net observation-probability-estimators are trained by a completely different algorithm known as **error back-propagation**.

GAUSSIAN

NEURAL NETWORKS

MULTI-LAYER PERCEPTRONS

ERROR BACK-PROPAGATION

In the simplest use of Gaussians, we assume that the possible values of the observation feature vector o_t are normally distributed, and so we represent the observation probability function $b_j(o_t)$ as a Gaussian curve with mean vector μ_j and covariance matrix $\sum_j$; (prime denotes vector transpose). We present the equation here for completeness, although we will not cover the details of the mathematics:

$$b_j(o_t) = \frac{1}{\sqrt{(2\pi)|\sum j|}} e^{[(o_t-\mu_j)'\Sigma_j^{-1}(o_t-\mu_j)]} \tag{7.10}$$

Usually we make the simplifying assumption that the covariance matrix Σ_j is diagonal, i.e., that it contains the simple variance of cepstral feature 1, the simple variance of cepstral feature 2, and so on, without worrying about the effect of cepstral feature 1 on the variance of cepstral feature 2. This means that in practice we are keeping only a single separate mean and variance for each feature in the feature vector.

Most recognizers do something even more complicated; they keep multiple Gaussians for each state, so that the probability of each feature of the observation vector is computed by adding together a variety of Gaussian curves. This technique is called **Gaussian mixtures**. In addition, many ASR systems share Gaussians between states in a technique known as **parameter tying** (or **tied mixtures**) (Huang and Jack, 1989). For example acoustically similar phone states might share (i.e., use the same) Gaussians for some features.

GAUSSIAN MIXTURES

TIED MIXTURES

How are the mean and covariance of the Gaussians estimated? It is helpful again to consider the simpler case of a non-hidden Markov Model, with only one state i. The vector of feature means μ and the vector of covariances Σ could then be estimated by averaging:

$$\hat{\mu}_i = \frac{1}{T}\sum_{t=1}^{T} o_t \tag{7.11}$$

$$\hat{\Sigma}_i = \frac{1}{T}\sum_{t=1}^{T}[(o_t-\mu_j)'(o_t-\mu_j)] \tag{7.12}$$

But since there are multiple hidden states, we don't know which observation vector o_t was produced by which state. Appendix D will show how the forward-backward algorithm can be modified to assign each observation vector o_t to every possible state i, prorated by the probability that the HMM was in state i at time t.

An alternative way to model continuous-valued features is the use of a **neural network**, **multilayer perceptron** (**MLP**) or **Artificial Neural Networks** (**ANNs**). Neural networks are far too complex for us to introduce in a page or two here; thus we will just give the intuition of how they are used in probability estimation as an alternative to Gaussian estimators. The interested reader should consult basic neural network textbooks (Anderson, 1995; Hertz et al., 1991) as well as references specifically focusing on neural-network speech recognition (Bourlard and Morgan, 1994).

NEURAL NETWORK

MULTILAYER PERCEPTRON

MLP

A neural network is a set of small computation units connected by weighted links. The network is given a vector of input values and computes

a vector of output values. The computation proceeds by each computational unit computing some non-linear function of its input units and passing the resulting value on to its output units.

The use of neural networks we will describe here is often called a **hybrid** HMM-MLP approach, since it uses some elements of the HMM (such as the state-graph representation of the pronunciation of a word) but the observation-probability computation is done by an MLP instead of a mixture of Gaussians. The input to these MLPs is a representation of the signal at a time t and some surrounding window; for example this might mean a vector of spectral features for a time t and eight additional vectors for times $t+10ms$, $t+20ms$, $t+30ms$, $t+40ms$, $t-10ms$, and so on. Thus the input to the network is a set of nine vectors, each vector having the complete set of real-valued spectral features for one time slice. The network has one output unit for each phone; by constraining the values of all the output units to sum to 1, the net can be used to compute the probability of a state j given an observation vector o_t, or $P(j|o_t)$. Figure 7.25 shows a sample of such a net.

HYBRID

This MLP computes the probability of the HMM state j given an observation o_t, or $P(q_j|o_t)$. But the observation likelihood we need for the HMM, $b_j(o_t)$, is $P(o_t|q_j)$. The Bayes rule can help us see how to compute one from the other. The net is computing:

$$p(q_j|o_t) = \frac{P(o_t|q_j)p(q_j)}{p(o_t)} \tag{7.13}$$

We can rearrange the terms as follows:

$$\frac{p(o_t|q_j)}{p(o_t)} = \frac{P(q_j|o_t)}{p(q_j)} \tag{7.14}$$

The two terms on the right-hand side of (7.14) can be directly computed from the MLP; the numerator is the output of the MLP, and the denominator is the total probability of a given state, summing over all observations (i.e., the sum over all t of $\sigma_j(t)$). Thus although we cannot directly compute $P(o_t|q_j)$, we *can* use (7.14) to compute $\frac{p(o_t|q_j)}{p(o_t)}$, which is known as a **scaled likelihood** (the likelihood divided by the probability of the observation). In fact, the scaled likelihood is just as good as the regular likelihood, since the probability of the observation $p(o_t)$ is a constant during recognition and doesn't hurt us to have in the equation.

SCALED LIKELIHOOD

The error-back-propagation algorithm for training an MLP requires that we know the correct phone label q_j for each observation o_t. Given a large training set of observations and correct labels, the algorithm iteratively adjusts the weights in the MLP to minimize the error with this training set.

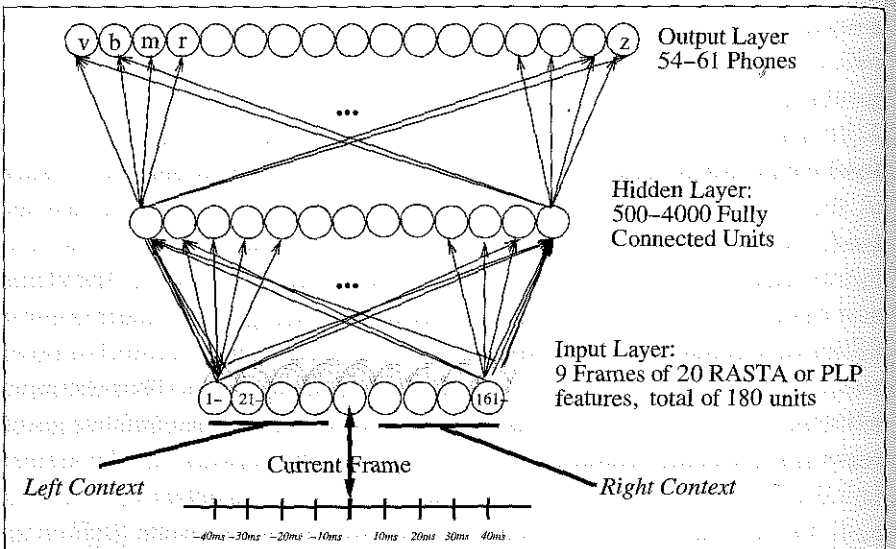


Figure 7.25 A neural net used to estimate phone state probabilities. Such a net can be used in an HMM model as an alternative to the Gaussian models. This particular net is from the MLP systems described in Bourlard and Morgan (1994); it is given a vector of features for a frame and for the four frames on either side, and estimates $p(q_j|o_t)$. This probability is then converted to an estimate of the observation likelihood $b = p(o_t|q_j)$ using the Bayes rule. These nets are trained using the error-back-propagation algorithm as part of the same **embedded training** algorithm that is used for Gaussians.

In the next section we will see where this labeled training set comes from, and how this training fits in with the **embedded training** algorithm used for HMMs. Neural nets seem to achieve roughly the same performance as a Gaussian model but have the advantage of using less parameters and the disadvantage of taking somewhat longer to train.

7.7 Training a Speech Recognizer

We have now introduced all the algorithms which make up the standard speech recognition system that was sketched in Figure 7.2 on page 241. We've seen how to build a Viterbi decoder, and how it takes 3 inputs (the observation likelihoods (via Gaussian or MLP estimation from the spectral features), the HMM lexicon, and the N-gram language model) and produces the most probable string of words. But we have not seen how all the proba-

METHODOLOGY BOX: WORD ERROR RATE

The standard evaluation metric for speech recognition systems is the **word error** rate. The word error rate is based on how much the word string returned by the recognizer (often called the **hypothesized** word string) differs from a correct or **reference** transcription. Given such a correct transcription, the first step in computing word error is to compute the **minimum edit distance** in words between the hypothesized and correct strings. The result of this computation will be the minimum number of word **substitutions**, word **insertions**, and word **deletions** necessary to map between the correct and hypothesized strings. The word error rate is then defined as follows (note that because the equation includes insertions, the error rate can be great than 100%):

$$\text{Word Error Rate} = 100\frac{\text{Insertions} + \text{Substitutions} + \text{Deletions}}{\text{Total Words in Correct Transcript}}$$

Here is an example of **alignments** between a reference and a hypothesized utterance from the CALLHOME corpus, showing the counts used to compute the word error rate:

REF:	i	***	**	UM	the	PHONE	IS	i	LEFT	THE	portable
HYP:	i	GOT	IT	TO	the	*****	FULLEST	i	LOVE	TO	portable
Eval:		I	I	S		D	S		S	S	

REF:	****	PHONE	UPSTAIRS	last	night	so	the	battery	ran	out	
HYP:	FORM	OF	STORES	last	night	so	the	battery	ran	out	
Eval:	I	S	S								

This utterance has six substitutions, three insertions, and one deletion:

$$\text{Word Error Rate} = 100\frac{6+3+1}{18} = 56\%$$

As of the time of this writing, state-of-the-art speech recognition systems were achieving around 20% word error rate on natural-speech tasks like the National Institute of Standards and Technology (NIST)'s Hub4 test set from the Broadcast News corpus (Chen et al., 1999), and around 40% word error rate on NIST's Hub5 test set from the combined Switchboard, Switchboard-II, and CALLHOME corpora (Hain et al., 1999).

bilistic models that make up a recognizer get trained.

EMBEDDED TRAINING

In this section we give a brief sketch of the **embedded training** procedure that is used by most ASR systems, whether based on Gaussians, MLPs, or even vector quantization. Some of the details of the algorithm (like the forward-backward algorithm for training HMM probabilities) have been removed to Appendix D.

Let's begin by summarizing the four probabilistic models we need to train in a basic speech recognition system:

- **language model probabilities:** $P(w_i|w_{i-1}w_{i-2})$
- **observation likelihoods:** $b_j(o_t)$
- **transition probabilities:** a_{ij}
- **pronunciation lexicon:** HMM state graph structure

In order to train these components we usually have

- a training corpus of speech wavefiles, together with a word-transcription
- a much larger corpus of text for training the language model, including the word-transcriptions from the speech corpus together with many other similar texts
- often a smaller training corpus of speech which is phonetically labeled (i.e., frames of the acoustic signal are hand-annotated with phonemes)

Let's begin with the N-gram language model. This is trained in the way we described in Chapter 6; by counting N-gram occurrences in a large corpus, then smoothing and normalizing the counts. The corpus used for training the language model is usually much larger than the corpus used to train the HMM a and b parameters. This is because the larger the training corpus the more accurate the models. Since N-gram models are much faster to train than HMM observation probabilities, and since text just takes less space than speech, it turns out to be feasible to train language models on huge corpora of as much as half a billion words of text. Generally the corpus used for training the HMM parameters is included as part of the language model training data; it is important that the acoustic and language model training be consistent.

The HMM lexicon structure is built by hand, by taking an off-the-shelf pronunciation dictionary such as the PRONLEX dictionary (LDC, 1995) or the CMUdict dictionary, both described in Chapter 4. In some systems, each phone in the dictionary maps into a state in the HMM. So the word *cat* would have three states corresponding to [k], [ae], and [t]. Many systems, however, use the more complex **subphone** structure described on page 251, in which

each phone is divided into 3 states: the beginning, middle and final portions of the phone, and in which furthermore there are separate instances of each of these subphones for each **triphone** context.

The details of the embedded training of the HMM parameters varies; we'll present a simplified version. First, we need some initial estimate of the transition and observation probabilities a_{ij} and $b_j(o_t)$. For the transition probabilities, we start by assuming that for any state all the possible following states are all equiprobable. The observation probabilities can be bootstrapped from a small hand-labeled training corpus. For example, the TIMIT or Switchboard corpora contain approximately 4 hours each of phonetically labeled speech. They supply a "correct" phone state label q for each frame of speech. These can be fed to an MLP or averaged to give initial Gaussian means and variances. For MLPs this initial estimate is important, and so a hand-labeled bootstrap is the norm. For Gaussian models the initial value of the parameters seems to be less important and so the initial mean and variances for Gaussians often are just set identically for all states by using the mean and variances of the entire training set.

Now we have initial estimates for the a and b probabilities. The next stage of the algorithm differs for Gaussian and MLP systems. For MLP systems we apply what is called a **forced Viterbi** alignment. A forced Viterbi alignment takes as input the correct words in an utterance, along with the spectral feature vectors. It produces the best sequence of HMM states, with each state aligned with the feature vectors. A forced Viterbi is thus a simplification of the regular Viterbi decoding algorithm, since it only has to figure out the correct phone sequence, but doesn't have to discover the word sequence. It is called **forced** because we constrain the algorithm by requiring the best path to go through a particular sequence of words. It still requires the Viterbi algorithm since words have multiple pronunciations, and since the duration of each phone is not fixed. The result of the forced Viterbi is a set of features vectors with "correct" phone labels, which can then be used to retrain the neural network. The counts of the transitions which are taken in the forced alignments can be used to estimate the HMM transition probabilities.

FORCED VITERBI

For the Gaussian HMMs, instead of using forced Viterbi, we use the forward-backward algorithm described in Appendix D. We compute the forward and backward probabilities for each sentence given the initial a and b probabilities, and use them to re-estimate the a and b probabilities. Just as for the MLP situation, the forward-backward algorithm needs to be constrained by our knowledge of the correct words. The forward-backward al-

gorithm computes its probabilities given a model λ. We use the "known" words sequence in a transcribed sentence to tell us which word models to string together to get the model λ that we use to compute the forward and backward probabilities for each sentence.

7.8 WAVEFORM GENERATION FOR SPEECH SYNTHESIS

Now that we have covered acoustic processing we can return to the acoustic component of a text-to-speech (TTS) system. Recall from Chapter 4 that the output of the linguistic processing component of a TTS system is a sequence of phones, each with a duration, and a F0 contour that specifies the pitch. This specification is often called the **target**, as it is this that we want the synthesizer to produce.

TARGET

The most commonly used type of algorithm works by **waveform concatenation**. Such **concatenative synthesis** is based on a database of speech that has been recorded by a single speaker. This database is then segmented into a number of short units, which can be phones, diphones, syllables, words or other units. The simplest sort of synthesizer would have phone units and the database would have a single unit for each phone in the phone inventory. By selecting units appropriately, we can generate a series of units which match the phone sequence in the input. By using signal processing to smooth joins at the unit edges, we can simply concatenate the waveforms for each of these units to form a single synthetic speech waveform.

WAVEFORM CONCATENATION

Experience has shown that single phone concatenative systems don't produce good quality speech. Just as in speech recognition, the context of the phone plays an important role in its acoustic pattern and hence a /t/ before a /a/ sounds very different from a /t/ before an /s/.

The triphone models described in Figure 7.11 on page 251 are a popular choice of unit in speech recognition, because they cover both the left and right contexts of a phone. Unfortunately, a language typically has a very large number of triphones (tens of thousands) and it is currently prohibitive to collect so many units for speech synthesis. Hence **diphones** are often used in speech synthesis as they provide a reasonable balance between context-dependency and size (typically 1000–2000 in a language). In speech synthesis, diphone units normally start half-way through the first phone and end half-way through the second. This is because it is known that phones are more stable in the middle than at the edges, so that the middles of most /a/ phones in a diphone are reasonably similar, even if the acoustic patterns start

DIPHONES

to differ substantially after that. If diphones are concatenated in the middles of phones, the discontinuities between adjacent units are often negligible.

Pitch and Duration Modification

The diphone synthesizer as just described will produce a reasonable quality speech waveform corresponding to the requested phone sequence. But the pitch and duration (i.e., the prosody) of each phone in the concatenated waveform will be the same as when the diphones were recorded and will not correspond to the pitch and durations requested in the input. The next stage of the synthesis process therefore is to use signal processing techniques to change the prosody of the concatenated waveform.

The linear prediction (LPC) model described earlier can be used for prosody modification as it explicitly separates the pitch of a signal from its spectral envelope If the concatenated waveform is represented by a sequence of linear prediction coefficients, a set of pulses can be generated corresponding to the desired pitch and used to re-excite the coefficients to produce a speech waveform again. By contracting and expanding frames of coefficients, the duration can be changed. While linear prediction produces the correct F0 and durations it produces a somewhat "buzzy" speech signal.

Another technique for achieving the same goal is the time-domain pitch-synchronous overlap and add (**TD-PSOLA**) technique. TD-PSOLA works **pitch-synchronously** in that each frame is centered around a **pitchmark** in the speech, rather than at regular intervals as in normal speech signal processing. The concatenated waveform is split into a number of frames, each centered around a pitchmark and extending a pitch period either side. Prosody is changed by recombining these frames at a new set of pitchmarks determined by the requested pitch and duration of the input. The synthetic waveform is created by simply overlapping and adding the frames. Pitch is increased by making the new pitchmarks closer together (shorter pitch periods implies higher frequency pitch), and decreased by making them further apart. Speech is made longer by duplication frames and shorter by leaving frames out. The operation of TD-PSOLA can be compared to that of a tape recorder with variable speed — if you play back a tape faster than it was recorded, the pitch periods will come closer together and hence the pitch will increase. But speeding up a tape recording effectively increases the frequency of *all* the components of the speech (including the formants which characterize the vowels) and will give the impression of a "squeaky", unnatural voice. TD-PSOLA differs because it separates each frame first and then

TD-PSOLA

decreases the distance between the frames. Because the internals of each frame aren't changed, the frequency of the non-pitch components is hardly altered, and the resultant speech sounds the same as the original except with a different pitch.

Unit Selection

While signal processing and diphone concatenation can produce reasonable quality speech, the result is not ideal. There are a number of reasons for this, but they all boil down to the fact that having a single example of each diphone is not enough. First of all, signal processing inevitably incurs distortion, and the quality of the speech gets worse when the signal processing has to stretch the pitch and duration by large amounts. Furthermore, there are many other subtle effects which are outside the scope of most signal processing algorithms. For instance, the amount of vocal effort decreases over time as the utterance is spoken, producing weaker speech at the end of the utterance. If diphones are taken from near the start of an utterance, they will sound unnatural in phrase-final positions.

Unit-selection synthesis is an attempt to address this problem by collecting several examples of each unit at different pitches and durations and linguistic situations, so that the unit is close to the target in the first place and hence the signal processing needs to do less work. One technique for unit-selection (Hunt and Black, 1996) works as follows:

The input to the algorithm is the same as other concatenative synthesizers, with the addition that the F0 contour is now specified as three F0 values per phone, rather than as a contour. The technique uses phones as its units, indexing phones in a large database of naturally occurring speech Each phone in the database is also marked with a duration and three pitch values. The algorithm works in two stages. First, for each phone in the target word, a set of candidate units which match closely in terms of phone identity, duration and F0 is selected from the database. These candidates are ranked using a **target cost** function, which specifies just how close each unit actually is to the target. The second part of the algorithm works by measuring how well each candidate for each unit joins with its neighbor's candidates. Various locations for the joins are assessed, which allows the potential for units to be joined in the middle, as with diphones. These potential joins are ranked using a **concatenation cost** function. The final step is to pick the best set of units which minimize the overall target and concatenation cost for the whole sentence. This step is performed using the Viterbi algorithm in a sim-

ilar way to HMM speech recognition: here the target cost is the observation probability and the concatenation cost is the transition probability.

By using a much larger database which contains many examples of each unit, unit-selection synthesis often produces more natural speech than straight diphone synthesis. Some systems then use signal processing to make sure the prosody matches the target, while others simply concatenate the units following the idea that a utterance which only roughly matches the target is better than one that exactly matches it but also has some signal processing distortion.

7.9 HUMAN SPEECH RECOGNITION

Speech recognition in humans shares some features with the automatic speech recognition models we have presented. We mentioned above that signal processing algorithms like PLP analysis (Hermansky, 1990) were in fact inspired by properties of the human auditory system. In addition, four properties of human **lexical access** (the process of retrieving a word from the mental lexicon) are also true of ASR models: **frequency**, **parallelism**, **neighborhood effects**, and **cue-based processing**. For example, as in ASR with its N-gram language models, human lexical access is sensitive to word **frequency**. High-frequency spoken words are accessed faster or with less information than low-frequency words. They are successfully recognized in noisier environments than low frequency words, or when only parts of the words are presented (Howes, 1957; Grosjean, 1980; Tyler, 1984, inter alia). Like ASR models, human lexical access is **parallel**: multiple words are active at the same time (Marslen-Wilson and Welsh, 1978; Salasoo and Pisoni, 1985, inter alia). Human lexical access exhibits **neighborhood effects** (the neighborhood of a word is the set of words which closely resemble it). Words with large frequency-weighted neighborhoods are accessed slower than words with less neighbors (Luce et al., 1990). Jurafsky (1996) shows that the effect of neighborhood on access can be explained by the Bayesian models used in ASR.

LEXICAL ACCESS

Finally, human speech perception is **cue based**: speech input is interpreted by integrating cues at many different levels. For example, there is evidence that human perception of individual phones is based on the integration of multiple cues, including acoustic cues, such as formant structure or the exact timing of voicing, (Oden and Massaro, 1978; Miller, 1994), visual cues, such as lip movement (Massaro and Cohen, 1983; Massaro, 1998),

and lexical cues such as the identity of the word in which the phone is placed (Warren, 1970; Samuel, 1981; Connine and Clifton, 1987; Connine, 1990). For example, in what is often called the **phoneme restoration effect**, Warren (1970) took a speech sample and replaced one phone (e.g. the [s] in *legislature*) with a cough. Warren found that subjects listening to the resulting tape typically heard the entire word *legislature* including the [s], and perceived the cough as background. Other cues in human speech perception include semantic **word association** (words are accessed more quickly if a semantically related word has been heard recently) and **repetition priming** (words are accessed more quickly if they themselves have just been heard). The intuitions of both these results are incorporated into recent language models discussed in Chapter 6, such as the cache model of Kuhn and de Mori (1990), which models repetition priming, or the trigger model of Rosenfeld (1996) and the LSA models of Coccaro and Jurafsky (1998) and Bellegarda (1999), which model word association. In a fascinating reminder that good ideas are never discovered only once, Cole and Rudnicky (1983) point out that many of these insights about context effects on word and phone processing were actually discovered by William Bagley (1901). Bagley achieved his results, including an early version of the phoneme restoration effect, by recording speech on Edison phonograph cylinders, modifying it, and presenting it to subjects. Bagley's results were forgotten and only rediscovered much later.[5]

WORD ASSOCIATION
REPETITION PRIMING

One difference between current ASR models and human speech recognition is the time-course of the model. It is important for the performance of the ASR algorithm that the the decoding search optimizes over the entire utterance. This means that the best sentence hypothesis returned by a decoder at the end of the sentence may be very different than the current-best hypothesis, halfway into the sentence. By contrast, there is extensive evidence that human processing is **on-line**: people incrementally segment and utterance into words and assign it an interpretation as they hear it. For example, Marslen-Wilson (1973) studied **close shadowers**: people who are able to shadow (repeat back) a passage as they hear it with lags as short as 250 ms. Marslen-Wilson found that when these shadowers made errors, they were syntactically and semantically appropriate with the context, indicating that word segmentation, parsing, and interpretation took place within these 250 ms. Cole (1973) and Cole and Jakimik (1980) found similar effects in their work on the detection of mispronunciations. These results have led psychological models of human speech perception (such as the Cohort model

ON-LINE

[5] Recall the discussion on page 15 of multiple independent discovery in science.

(Marslen-Wilson and Welsh, 1978) and the computational TRACE model (McClelland and Elman, 1986)) to focus on the time-course of word selection and segmentation. The TRACE model, for example, is a **connectionist** or **neural network** interactive-activation model, based on independent computational units organized into three levels: feature, phoneme, and word. Each unit represents a hypothesis about its presence in the input. Units are activated in parallel by the input, and activation flows between units; connections between units on different levels are excitatory, while connections between units on single level are inhibitatory. Thus the activation of a word slightly inhibits all other words.

CONNECTIONIST
NEURAL NETWORK

We have focused on the similarities between human and machine speech recognition; there are also many differences. In particular, many other cues have been shown to play a role in human speech recognition but have yet to be successfully integrated into ASR. The most important class of these missing cues is prosody. To give only one example, Cutler and Norris (1988), Cutler and Carter (1987) note that most multisyllabic English word tokens have stress on the initial syllable, suggesting in their metrical segmentation strategy (MSS) that stress should be used as a cue for word segmentation.

7.10 SUMMARY

Together with Chapters 4–6, this chapter introduced the fundamental algorithms for addressing the problem of **Large Vocabulary Continuous Speech Recognition** and **Text-To-Speech synthesis**.

- The input to a speech recognizer is a series of acoustic waves. The **waveform**, **spectrogram** and **spectrum** are among the visualization tools used to understand the information in the signal.
- In the first step in speech recognition, wound waves are **sampled**, **quantized**, and converted to some sort of **spectral representation**; A commonly used spectral representation is the **LPC cepstrum**, which provides a vector of features for each time-slice of the input.
- These **feature vectors** are used to estimate the **phonetic likelihoods** (also called **observation likelihoods**) either by a mixture of **Gaussian** estimators or by a **neural net**.
- **Decoding** or **search** is the process of finding the optimal sequence of model states which matches a sequence of input observations. (The

fact that are two terms for this process is a hint that speech recognition is inherently inter-disciplinary, and draws its metaphors from more than one field; **decoding** comes from information theory, and **search** from artificial intelligence).

- We introduced two decoding algorithms: time-synchronous **Viterbi** decoding (which is usually implemented with pruning and can then be called **beam search**) and **stack** or $\mathbf{A}^*$ decoding. Both algorithms take as input a series of feature vectors, and two ancillary algorithms: one for assigning likelihoods (e.g., Gaussians or MLP) and one for assigning priors (e.g., an N-gram language model). Both give as output a string of words.
- The **embedded training** paradigm is the normal method for training speech recognizers. Given an initial lexicon with hand-built pronunciation structures, it will train the HMM transition probabilities and the HMM observation probabilities. This HMM observation probability estimation can be done via a Gaussian or an MLP.
- One way to implement the acoustic component of a TTS system is with **concatenative synthesis**, in which an utterance is built by concatenating and then smoothing diphones taken from a large database of speech recorded by a single speaker.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The first machine which recognized speech was probably a commercial toy named "Radio Rex" which was sold in the 1920s. Rex was a celluloid dog that moved (via a spring) when the spring was released by 500 Hz acoustic energy. Since 500 Hz is roughly the first formant of the vowel in "Rex", the dog seemed to come when he was called (David and Selfridge, 1962).

By the late 1940s and early 1950s, a number of machine speech recognition systems had been built. An early Bell Labs system could recognize any of the 10 digits from a single speaker (Davis et al., 1952). This system had 10 speaker-dependent stored patterns, one for each digit, each of which roughly represented the first two vowel formants in the digit. They achieved 97–99% accuracy by choosing the pattern which had the highest relative correlation coefficient with the input. Fry (1959) and Denes (1959) built a phoneme recognizer at University College, London, which recognized four vowels and nine consonants based on a similar pattern-recognition principle.

Fry and Denes's system was the first to use phoneme transition probabilities to constrain the recognizer.

The late 1960s and early 1970s produced a number of important paradigm shifts. First were a number of feature-extraction algorithms, include the efficient Fast Fourier Transform (FFT) (Cooley and Tukey, 1965), the application of cepstral processing to speech (Oppenheim et al., 1968), and the development of LPC for speech coding (Atal and Hanauer, 1971). Second were a number of ways of handling **warping**; stretching or shrinking the input signal to handle differences in speaking rate and segment length when matching against stored patterns. The natural algorithm for solving this problem was dynamic programming, and, as we saw in Chapter 5, the algorithm was reinvented multiple times to address this problem. The first application to speech processing was by Vintsyuk (1968), although his result was not picked up by other researchers, and was reinvented by Velichko and Zagoruyko (1970) and Sakoe and Chiba (1971) (and (1984)). Soon afterwards, Itakura (1975) combined this dynamic programming idea with the LPC coefficients that had previously been used only for speech coding. The resulting system extracted LPC features for incoming words and used dynamic programming to match them against stored LPC templates.

WARPING

The third innovation of this period was the rise of the HMM. Hidden Markov Models seem to have been applied to speech independently at two laboratories around 1972. One application arose from the work of statisticians, in particular Baum and colleagues at the Institute for Defense Analyses in Princeton on HMMs and their application to various prediction problems (Baum and Petrie, 1966; Baum and Eagon, 1967). James Baker learned of this work and applied the algorithm to speech processing (Baker, 1975) during his graduate work at CMU. Independently, Frederick Jelinek, Robert Mercer, and Lalit Bahl (drawing from their research in information-theoretical models influenced by the work of Shannon (1948)) applied HMMs to speech at the IBM Thomas J. Watson Research Center (Jelinek et al., 1975). IBM's and Baker's systems were very similar, particularly in their use of the Bayesian framework described in this chapter. One early difference was the decoding algorithm; Baker's DRAGON system used Viterbi (dynamic programming) decoding, while the IBM system applied Jelinek's stack decoding algorithm (Jelinek, 1969). Baker then joined the IBM group for a brief time before founding the speech-recognition company Dragon Systems. The HMM approach to speech recognition would turn out to completely dominate the field by the end of the century; indeed the IBM lab was the driving force in extending statistical models to natu-

ral language processing as well, including the development of class-based N-grams, HMM-based part-of-speech tagging, statistical machine translation, and the use of entropy/perplexity as an evaluation metric.

The use of the HMM slowly spread through the speech community. One cause was a number of research and development programs sponsored by the Advanced Research Projects Agency of the U.S. Department of Defense (ARPA). The first five-year program starting in 1971, and is reviewed in Klatt (1977). The goal of this first program was to build speech understanding systems based on a few speakers, a constrained grammar and lexicon (1000 words), and less than 10% semantic error rate. Four systems were funded and compared against each other: the System Development Corporation (SDC) system, Bolt, Beranek & Newman (BBN)'s HWIM system, Carnegie-Mellon University's Hearsay-II system, and Carnegie-Mellon's Harpy system (Lowerre, 1968). The Harpy system used a simplified version of Baker's HMM-based DRAGON system and was the best of the tested systems, and according to Klatt the only one to meet the original goals of the ARPA project (with a semantic error rate of 94% on a simple task).

Beginning in the mid-1980s, ARPA funded a number of new speech research programs. The first was the "Resource Management" (RM) task (Price et al., 1988), which like the earlier ARPA task involved transcription (recognition) of read-speech (speakers reading sentences constructed from a 1000-word vocabulary) but which now included a component that involved speaker-independent recognition. Later tasks included recognition of sentences read from the Wall Street Journal (WSJ) beginning with limited systems of 5,000 words, and finally with systems of unlimited vocabulary (in practice most systems use approximately 60,000 words). Later speech-recognition tasks moved away from read-speech to more natural domains; the Broadcast News (also called Hub-4) domain (LDC, 1998; Graff, 1997) (transcription of actual news broadcasts, including quite difficult passages such as on-the-street interviews) and the CALLHOME and CALLFRIEND domain (LDC, 1999) (natural telephone conversations between friends), part of what was also called Hub-5. The Air Traffic Information System (ATIS) task (Hemphill et al., 1990) was a speech understanding task whose goal was to simulate helping a user book a flight, by answering questions about potential airlines, times, dates, and so forth.

BAKE-OFF

Each of the ARPA tasks involved an approximately annual **bake-off** at which all ARPA-funded systems, and many other 'volunteer' systems from North American and Europe, were evaluated against each other in terms of word error rate or semantic error rate. In the early evaluations, for-profit cor-

porations did not generally compete, but eventually many (especially IBM and ATT) competed regularly. The ARPA competitions resulted in widescale borrowing of techniques among labs, since it was easy to see which ideas had provided an error-reduction the previous year, and were probably an important factor in the eventual spread of the HMM paradigm to virtual every major speech recognition lab. The ARPA program also resulted in a number of useful databases, originally designed for training and testing systems for each evaluation (TIMIT, RM, WSJ, ATIS, BN, CALLHOME, Switchboard) but then made available for general research use.

There are a number of textbooks on speech recognition that are good choices for readers who seek a more in-depth understanding of the material in this chapter: Jelinek (1997), Gold and Morgan (1999), and Rabiner and Juang (1993) are the most comprehensive. The last two textbooks also have comprehensive discussions of the history of the field, and together with the survey paper of Levinson (1995) have influenced our short history discussion in this chapter. Our description of the forward-backward algorithm was modeled after Rabiner (1989). Another useful tutorial paper is Knill and Young (1997). Research in the speech recognition field often appears in the proceedings of the biennial EUROSPEECH Conference and the International Conference on Spoken Language Processing (ICSLP), held in alternating years, as well as the annual IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP). Journals include Speech Communication, Computer Speech and Language, IEEE Transactions on Pattern Analysis and Machine Intelligence, and IEEE Transactions on Acoustics, Speech, and Signal Processing.

EXERCISES

7.1 Analyze each of the errors in the incorrectly recognized transcription of "um the phone is I left the..." on page 271. For each one, give your best guess as to whether you think it is caused by a problem in signal processing, pronunciation modeling, lexicon size, language model, or pruning in the decoding search.

7.2 In practice, speech recognizers do all their probability computation using the **log probability** (or **logprob**) rather than actual probabilities. This LOGPROB

helps avoid underflow for very small probabilities, but also makes the Viterbi algorithm very efficient, since all probability multiplications can be implemented by adding log probabilities. Rewrite the pseudocode for the Viterbi algorithm in Figure 7.9 on page 249 to make use of logprobs instead of probabilities.

7.3 Now modify the Viterbi algorithm in Figure 7.9 on page 249 to implement the beam search described on page 251. Hint: You will probably need to add in code to check whether a given state is at the end of a word or not.

7.4 Finally, modify the Viterbi algorithm in Figure 7.9 on page 249 with more detailed pseudocode implementing the array of backtrace pointers.

7.5 Implement the Stack decoding algorithm of Figure 7.14 on 256. Pick a very simple h^* function like an estimate of the number of words remaining in the sentence.

7.6 Modify the forward algorithm of Figure 5.16 to use the tree-structured lexicon of Figure 7.18 on page 259.

17 WORD SENSE DISAMBIGUATION AND INFORMATION RETRIEVAL

Oh are you from Wales?
Do you know a fella named Jonah?
He used to live in whales for a while.
Groucho Marx

This chapter introduces a number of topics related to **lexical semantic processing**. By this, we have in mind applications that make use of word meanings, but which are to varying degrees decoupled from the more complex tasks of compositional sentence analysis and discourse understanding.

LEXICAL SEMANTIC PROCESSING

The first topic we cover, **word sense disambiguation**, is of considerable theoretical and practical interest. Recall from Chapter 16 that the task of word sense disambiguation is to examine word tokens in context and specify exactly which sense of each word is being used. As we will see, this is a non-trivial undertaking given the somewhat illusive nature of a word sense. Nevertheless, there are robust algorithms that can achieve high levels of accuracy given certain reasonable assumptions.

WORD SENSE DISAMBIGUATION

The second topic we cover, **information retrieval**, is an extremely broad field, encompassing a wide-range of topics pertaining to the storage, analysis, and retrieval of all manner of media (Baeza-Yates and Ribeiro-Neto, 1999). Our concern in this chapter is solely with the storage and retrieval of text documents in response to users' requests for information. We are interested in approaches in which users' needs are expressed as words, and documents are represented in terms of the words they contain. Section 17.3 presents the **vector space model**, some variant of which is used in many current systems, including most Web search engines.

INFORMATION RETRIEVAL

17.1 SELECTIONAL RESTRICTION-BASED DISAMBIGUATION

For the most part, our discussions of compositional semantic analyzers in Chapter 15 ignored the issue of lexical ambiguity. By now it should be clear that this is not a reasonable approach. Without some means of selecting correct senses for the words in the input, the enormous amount of homonymy and polysemy in the lexicon will quickly overwhelm any approach in an avalanche of competing interpretations. As with syntactic part-of-speech tagging, there are two fundamental approaches to handling this ambiguity problem. In an integrated rule-to-rule approach to semantic analysis, the selection of correct word senses occurs during semantic analysis as a side-effect of the elimination of ill-formed semantic representations. In a stand-alone approach, sense disambiguation is performed independent of, and prior to, compositional semantic analysis. This section discusses the role of selectional restrictions in the former approach. The stand-alone approach is discussed in detail in Section 17.2.

Selectional restrictions and type hierarchies are the primary knowledge-sources used to perform disambiguation in most integrated approaches. They are used to rule out inappropriate senses and thereby reduce the amount of ambiguity present during semantic analysis. In an integrated rule-to-rule approach to semantic analysis, selectional restrictions are used to block the formation of component meaning representations that contain selectional restriction violations. By blocking such ill-formed components, the semantic analyzer will find itself dealing with fewer ambiguous meaning representations. This ability to focus on correct senses by eliminating flawed representations that result from incorrect senses can be viewed as a form of indirect word sense disambiguation. While the linguistic basis for this approach can be traced back to the work of Katz and Fodor (1963), the most sophisticated computational exploration of it is due to Hirst (1987).

As an example of this approach, consider the following pair of WSJ examples, focusing solely on their use of the lexeme *dish*:

(17.1) "In our house, everybody has a career and none of them includes washing **dishes**," he says.

(17.2) In her tiny kitchen at home, Ms. Chen works efficiently, stir-frying several simple **dishes**, including braised pig's ears and chicken livers with green peppers.

These examples make use of two polysemous senses of the lexeme *dish*. The first refers to the physical objects that we eat from, while the second refers to

the actual meals or recipes. The fact that we perceive no ambiguity in these examples can be attributed to the selectional restrictions imposed by *wash* and *stir-fry* on their PATIENT roles, along with the semantic type information associated with the two senses of *dish*. The restrictions imposed by *wash* conflict with the food sense of dish since it does not denote something that is normally washable. Similarly, the restrictions on *stir-fry* conflict with the artifact sense of dish, since it does not denote something edible. Therefore, in both of these cases *the predicate selects the correct sense* of an ambiguous argument by eliminating the sense that fails to match one of its selectional restrictions.

Now consider the following WSJ and ATIS examples, focusing on the ambiguous predicate *serve*:

(17.3) Well, there was the time they **served** green-lipped mussels from New Zealand.

(17.4) Which airlines **serve** Denver?

(17.5) Which ones **serve** breakfast?

Here the sense of *serve* in example (17.3) requires some kind of food as its PATIENT, the sense in example (17.4) requires some kind of geographical or political entity, and the sense in the last example requires a meal designator. If we assume that *mussels*, *Denver* and *breakfast* are unambiguous, then it is the arguments in these examples that select the appropriate sense of the verb.

Of course, there are also cases where both the predicate and the argument have multiple senses. Consider the following BERP example:

(17.6) I'm looking for a restaurant that **serves** vegetarian **dishes**.

Restricting ourselves to three senses of *serve* and two senses of *dish* yields six possible sense combinations in this example. However, since only one combination of the six is free from a selectional restriction violation, determining the correct sense of both *serve* and *dish* is straightforward; the predicate and argument mutually select the correct senses.

Although there are a wide variety of ways to integrate this style of disambiguation into a semantic analyzer, the most straightforward approach follows the rule-to-rule strategy introduced in Chapter 15. In this integrated approach, fragments of meaning representations are composed and checked for selectional restriction violations as soon as their corresponding syntactic constituents are created. Those representations that contain selectional restriction violations are eliminated from further consideration.

This approach requires two additions to the knowledge structures used in semantic analyzers: access to hierarchical type information about arguments, and semantic selectional restriction information about the arguments to predicates. Recall from Chapter 16 that both of these can be encoded using knowledge from WordNet. The type information is available in the form of the hypernym information about the heads of the meaning structures being used as arguments to predicates. The selectional restriction information about argument roles can be encoded by associating the appropriate WordNet synsets with the arguments to each predicate-bearing lexical item.

Limitations of Selectional Restrictions

There are a number of practical and theoretical problems with this use of selectional restrictions. The first symptom of these problems is the fact that there are examples like the following where the available selectional restrictions are too general to uniquely select a correct sense:

(17.7) What kind of **dishes** do you recommend?

In cases like this, we either have to rely on the stand-alone methods to be discussed in Section 17.2, or knowledge of the broader discourse context, as will be discussed in Chapter 18.

More problematic are examples that contain obvious violations of selectional restrictions but are nevertheless perfectly well-formed and interpretable. Therefore, any approach based on a strict *elimination* of such interpretations is in serious trouble. Consider the following WSJ example:

(17.8) But it fell apart in 1931, perhaps because people realized you can't **eat** gold for lunch if you're hungry.

The phrase *eat gold* clearly violates the selectional restriction that *eat* places on its PATIENT role. Nevertheless, this example is perfectly well-formed. The key is the negative environment set up by *can't* prior to the violation of the restriction. This example makes it clear that any purely local, or rule-to-rule, analysis of selectional restrictions will fail when a wider context makes the violation of a selectional restriction acceptable.

A second problem with selectional restrictions is illustrated by the following example:

(17.9) In his two championship trials, Mr. Kulkarni **ate** glass on an empty stomach, accompanied only by water and tea.

Although the event described in this example is somewhat unusual, the sentence itself is not semantically ill-formed, despite the violation of *eat*'s selec-

tional restriction. Examples such as this illustrate the fact that thematic roles and selectional restrictions are merely loose approximations of the deeper concepts they represent. They cannot hope to account for uses that require deeper commonsense knowledge about what eating is all about. At best, they reflect the idea that the things that are eaten are normally edible.

Finally, as discussed in Chapter 16, metaphoric and metonymic uses challenge this approach as well. Consider the following WSJ example:

(17.10) If you want to **kill** the Soviet Union, get it to try to **eat** Afghanistan.

Here the typical selectional restrictions on the PATIENTS of both *kill* and *eat* will eliminate all possible literal senses leaving the system with no possible meanings. In many systems, such a situation serves to trigger alternative mechanisms for interpreting metaphor and metonymy (Fass, 1997).

As Hirst (1987) observes, examples like these often result in the elimination of all senses, bringing semantic analysis to a halt. One approach to alleviating this problem is to adopt the view of selectional restrictions as preferences, rather than rigid requirements. Although there have been many instantiations of this approach over the years (Wilks, 1975c, 1975b, 1978), the one that has received the most thorough empirical evaluation is Resnik's (1997) work, which uses the notion of a **selectional association**. A selectional association is a probabilistic measure of the strength of association between a predicate and a class dominating the argument to the predicate. Resnik (1997) gives a method for deriving these associations using WordNet's hyponymy relations combined with a tagged corpus containing verb-argument relations.

Resnik (1998) shows that these selectional associations can be used to perform a limited form of word sense disambiguation. Roughly speaking the algorithm selects as the correct sense for an argument, the one that has the highest selectional association between one of its ancestor hypernyms and the predicate. Resnik (1997) reports an average of 44% correct with this technique for verb-object relationships, a result that is an improvement over the most frequent sense baseline which performs at 28%. A limitation of this approach is that it only addresses the case where the predicate is unambiguous and *selects* the correct sense of the argument. A more complex decision criteria would be needed for the situation where both the predicate and argument are ambiguous.

17.2 Robust Word Sense Disambiguation

The selectional restriction approach to disambiguation has too many requirements to be useful in large-scale practical applications. Even with the use of WordNet, the requirements of complete selectional restriction information for all predicate roles, and complete type information for the senses of all possible fillers are unlikely to be met. In addition, as we saw in Chapters 10, 12, and 15, the availability of a complete and accurate parse for all inputs is unlikely to be met in environments involving unrestricted text.

To address these concerns, a number of robust stand-alone disambiguation systems with more modest requirements have been developed over the years. As with part-of-speech taggers, these systems are designed to operate in a stand-alone fashion and make minimal assumptions about what information will be available from other processes. The following sections explore the application of supervised, bootstrapping, and unsupervised machine learning approaches to this problem. We then consider the role of machine readable dictionaries in the construction of stand-alone taggers.

Machine Learning Approaches

In machine learning approaches, systems are *trained* to perform the task of word sense disambiguation. In these approaches, what is learned is a classifier that can be used to assign as yet unseen examples to one of a fixed number of senses. As we will see, these approaches vary as to the nature of the training material, how much material is needed, the degree of human intervention, the kind of linguistic knowledge used, and the output produced. What they all share is an emphasis on acquiring the knowledge needed for the task from data, rather than from human analysts. The principal question to keep in mind as we explore these systems is whether the method scales; that is, would it be possible to apply the method to a substantial part of the entire vocabulary of a language?

The Inputs: Feature Vectors

In most of these approaches, the initial input consists of the word to be disambiguated, which we will refer to as the **target** word, along with a portion of the text in which it is embedded, which we will call its **context**. This initial input is then processed in the following ways:

- The input is normally part-of-speech tagged using one of the high accuracy methods described in Chapter 8.
- The original context may be replaced with larger or smaller segments surrounding the target word.
- Often some amount of stemming, or more sophisticated morphological processing, is performed on all the words in the context.
- Less often, some form of partial parsing, or dependency parsing, is performed to ascertain thematic or grammatical roles and relations.

After this initial processing, the input is then boiled down to a fixed set of features that capture information relevant to the learning task. This task consists of two steps: selecting the relevant linguistic features, and encoding them in a form usable in a learning algorithm. A simple **feature vector** consisting of numeric or nominal values can easily encode the most frequently used linguistic information, and is appropriate for use in most learning algorithms.

FEATURE VECTOR

The linguistic features used in training WSD systems can be roughly divided into two classes: collocational features and co-occurrence features. In general, the term **collocation** refers to a quantifiable position-specific relationship between two lexical items. Collocational features encode information about the lexical inhabitants of *specific* positions located to the left or right of the target word. Typical features include the word, the root form of the word, and the word's part-of-speech. Such features are effective at encoding local lexical and grammatical information that can often accurately isolate a given sense.

COLLOCATION

As an example of this type of feature-encoding, consider the situation where we need to disambiguate the word *bass* in the following example:

(17.11) An electric guitar and **bass** player stand off to one side, not really part of the scene, just as a sort of nod to gringo expectations perhaps.

A feature-vector consisting of the two words to the right and left of the target word, along with their respective parts-of-speech, would yield the following vector:

```
[guitar, NN1, and, CJC, player, NN1, stand, VVB]
```

The second type of feature consists of co-occurrence data about neighboring words, ignoring their exact position. In this approach, the words themselves (or their roots) serve as features. The value of the feature is the number of times the word occurs in a region surrounding the target word.

This region is most often defined as a fixed size window with the target word at the center. To make this approach manageable, a small number of frequently used content words are selected for use as features. This kind of feature is effective at capturing the general topic of the discourse in which the target word has occurred. This, in turn, tends to identify senses of a word that are specific to certain domains.

For example, a co-occurrence vector consisting of the 12 most frequent content words from a collection of *bass* sentences drawn from the WSJ corpus would have the following words as features: *fishing, big, sound, player, fly, rod, pound, double, runs, playing, guitar, band.* Using these words as features with a window size of 10, example (17.11) would be represented by the following vector:

```
[0,0,0,1,0,0,0,0,0,0,1,0]
```

As we will see, most robust approaches to sense disambiguation make use of a combination of both collocational and co-occurrence features.

Supervised Learning Approaches

SUPERVISED LEARNING

In supervised approaches, a sense disambiguation system is learned from a representative set of labeled instances drawn from the same distribution as the test set to be used. This is an application of the **supervised learning** approach to creating a classifier. In such approaches, a learning system is presented with a training set consisting of feature-encoded inputs *along with their appropriate label, or category.* The output of the system is a classifier system capable of assigning labels to new feature-encoded inputs.

Bayesian classifiers (Duda and Hart, 1973), decision lists (Rivest, 1987), decision trees (Quinlan, 1986), neural networks (Rumelhart et al., 1986), logic learning systems (Mooney, 1995), and nearest neighbor methods (Cover and Hart, 1967) all fit into this paradigm. We will restrict our discussion to the naive Bayes and decision list approaches, since they have been the focus of considerable work in word sense disambiguation.

NAIVE BAYES CLASSIFIER

The **naive Bayes classifier** approach to WSD is based on the premise that choosing the best sense for an input vector amounts to choosing the most probable sense given that vector. In other words:

$$\hat{s} = \underset{s \in S}{\operatorname{argmax}} P(s|V) \tag{17.12}$$

In this formula, S denotes the set of senses appropriate for the target associated with this vector, s denotes each of the possible senses in S, and V stands for the vector representation of the input context. As is almost always

METHODOLOGY BOX: EVALUATING WSD SYSTEMS

The basic metric used in evaluating sense disambiguation systems is simple precision: the percentage of words that are tagged correctly. The primary baseline against which this metric is compared is the **most frequent sense** metric (Gale et al., 1992): how well a system would perform if it simply chose the most frequent sense of a word.

The use of precision requires access to the correct senses for the words in a test set. Fortunately, two large sense-tagged corpora are now available: the SEMCOR corpus (Landes et al., 1998), which consists of a portion of the Brown corpus tagged with WordNet senses, and the SENSEVAL corpus (Kilgarriff and Rosenzweig, 2000), which is a tagged corpus derived from the HECTOR corpus and dictionary project.

One complication arising from the use of simple precision is that the nature of the senses used in an evaluation has a huge effect on the results. In particular, results derived from the use of coarse distinctions among homographs, such as the musical and fish senses of *bass*, can not easily be compared to results based on the use of fine-grained sense distinctions such as those found in traditional dictionaries, or lexical resources like WordNet.

A second complication has to do with metrics that go beyond simple precision and make use of **partial credit**. For example, confusing a particular musical sense of *bass* with a fish sense, is clearly worse than confusing it with another musical sense. With such a metric, an exact sense-match would receive full credit, while selecting a broader sense would receive partial credit. Of course, this kind of scheme is entirely dependent on the organization of senses in the particular dictionary being used.

Standardized evaluation frameworks for word sense disambiguation systems are now available. In particular, the SENSEVAL effort (Kilgarriff and Palmer, 2000), provides the same kind of evaluation framework for sense disambiguation, that the MUC (Sundheim, 1995b) and TREC (Voorhees and Harman, 1998) evaluations have provided for information extraction and information retrieval.

the case, it would be difficult to collect statistics for this equation directly. Instead, we rewrite it in the usual Bayesian manner as follows:

$$\hat{s} = \underset{s \in S}{\text{argmax}} \frac{P(V|s)P(s)}{P(V)} \tag{17.13}$$

Of course, the data available that associates specific vectors with senses is too sparse to be useful. However, what is available in abundance in a tagged training set is information about individual feature-value pairs in the context of specific senses. Therefore, we can make the independence assumption that gives this method its name, and that has served us well in part-of-speech tagging, speech recognition, and probabilistic parsing — naively assuming that the features are independent of one another. Making this assumption yields the following approximation for $P(V|s)$:

$$P(V|s) \approx \prod_{j=1}^{n} P(v_j|s) \tag{17.14}$$

In other words, we can estimate the probability of an entire vector given a sense by the product of the probabilities of its individual features given that sense.

Given this equation, **training** a naive Bayes classifier amounts to collecting counts of the individual feature-value statistics with respect to each sense of the target word in a sense-tagged training corpus. To make this concrete, let's return to example (17.11). The individual statistics needed for this example might include the probability of the word *player* occurring immediately to the right of a use of each of the *bass* senses, or the probability of the word *guitar* one place to the left of a use of one of the *bass* senses.

Returning to equation (17.13), the term $P(s)$ is the prior for each sense, which just corresponds to the proportion of each sense in the sense-tagged training corpus. Finally, since $P(V)$ is the same for all possible senses, it does not effect the final ranking of senses, leaving us with the following:

$$\hat{s} = \underset{s \in S}{\text{argmax}} P(s) \prod_{j=1}^{n} P(v_j|s) \tag{17.15}$$

Of course, all the issues discussed in Chapter 6 with respect to zero counts and smoothing apply here as well.

In a large experiment evaluating a number of supervised learning algorithms, Mooney (1996) reports that a naive-Bayes classifier and a neural network achieved the highest performance, both achieving around 73% correct in assigning one of six senses to a corpus of examples of the word *line*.

DECISION LIST CLASSIFIERS

Decision list classifiers are equivalent to simple case statements in

Rule		Sense
fish within window	$\Rightarrow$	**bass**^{1}
striped bass	$\Rightarrow$	**bass**^{1}
guitar within window	$\Rightarrow$	**bass**^{2}
bass player	$\Rightarrow$	**bass**^{2}
piano within window	$\Rightarrow$	**bass**^{2}
tenor within window	$\Rightarrow$	**bass**^{2}
sea bass	$\Rightarrow$	**bass**^{1}
play/V *bass*	$\Rightarrow$	**bass**^{2}
river within window	$\Rightarrow$	**bass**^{1}
violin within window	$\Rightarrow$	**bass**^{2}
salmon within window	$\Rightarrow$	**bass**^{1}
on bass	$\Rightarrow$	**bass**^{2}
bass are	$\Rightarrow$	**bass**^{1}

Figure 17.1 An abbreviated decision list for disambiguating the fish sense of bass from the music sense. Adapted from Yarowsky (1996).

most programming languages. In a decision list classifier, a sequence of tests is applied to each vector encoded input. If a test succeeds, then the sense associated with that test is returned. If the test fails, then the next test in the sequence is applied. This continues until the end of the list, where a default test simply returns the majority sense.

Figure 17.1 shows a portion of a decision list for the task of discriminating the fish sense of *bass* from the music sense. The first test says that if the word *fish* occurs anywhere within the input context then **bass**^{1} is the correct answer. If it doesn't then each of the subsequent tests is consulted in turn until one returns true; as with case statements a default test that returns true is included at the end of the list.

Learning a decision list classifier consists of generating and ordering individual tests based on the characteristics of the training data. There are a wide number of methods that can be used to create such lists. In the approach used by Yarowsky (1994) every individual feature-value pair constitutes a test. These tests are then ordered according to their individual accuracy on the entire training set, where the accuracy of a test is based on its log-likelihood ratio:

$$\text{Abs}(\text{Log}\left(\frac{P(Sense_1|f_i = v_j)}{P(Sense_2|f_i = v_j)}\right)) \tag{17.16}$$

The decision list is created from these tests by simply ordering the tests in the

list according to this measure, with each test returning the appropriate sense. Yarowsky (1996) reports that this technique consistently achieves over 95% correct on a wide variety of binary decision tasks.

We should note that this training method differs quite a bit from standard decision list learning algorithms. For the details and theoretical motivation for these approaches see Rivest (1987) or Russell and Norvig (1995).

Bootstrapping Approaches

BOOTSTRAPPING APPROACH

A major problem with supervised approaches is the need for a large sense-tagged training set. The **bootstrapping approach** (Hearst, 1991; Yarowsky, 1995) eliminates the need for a large training set by relying on a relatively small number of instances of each sense for each lexeme of interest. These labeled instances are used as **seeds** to train an initial classifier using any of the supervised learning methods mentioned in the last section. This initial classifier is then be used to extract a larger training set from the remaining untagged corpus. Repeating this process results in a series of classifiers with improving accuracy and coverage.

The key to this approach lies in its ability to create a larger training set from a small set of seeds. To succeed, it must include only those instances in which the initial classifier has a high degree of confidence. This larger training set is then used to create a new more accurate classifier with broader coverage. With each iteration of this process, the training corpus grows and the untagged corpus shrinks. As with most iterative methods, this process can be repeated until some sufficiently low error-rate on the training set is reached, or until no further examples from the untagged corpus are above threshold.

The initial seeds used in these bootstrapping methods can be generated in a number of ways. Hearst (1991) generates a seed set by simply hand-labeling a small set of examples from the initial corpus. This approach has three major advantages:

- There is a reasonable certainty that the seed instances are correct, thus ensuring that the learner does not get off on the wrong foot.
- The analyst can make some effort to choose examples that are not only correct, but in some sense prototypical of each sense.
- It is reasonably easy to carry out.

An effective alternative technique is to search for sentences containing words or phrases that are strongly associated with the target senses. Yarowsky (1995) calls this the **One Sense per Collocation** constraint and

Klucevsek **play**s Giulietti or Titano piano accordions with the more flexible, more difficult free **bass** rather than the traditional Stradella **bass** with its preset chords designed mainly for accompaniment.

We need more good teachers – right now, there are only a half a dozen who can **play** the free **bass** with ease.

An electric guitar and **bass play**er stand off to one side, not really part of the scene, just as a sort of nod to gringo expectations perhaps.

When the New Jersey Jazz Society, in a fund-raiser for the American Jazz Hall of Fame, honors this historic night next Saturday, Harry Goodman, Mr. Goodman's brother and **bass play**er at the original concert, will be in the audience with other family members.

The researchers said the worms spend part of their life cycle in such **fish** as Pacific salmon and striped **bass** and Pacific rockfish or snapper.

Associates describe Mr. Whitacre as a quiet, disciplined and assertive manager whose favorite form of escape is **bass fish**ing.

And it all started when **fish**ermen decided the striped **bass** in Lake Mead were too skinny.

Though still a far cry from the lake's record 52-pound **bass** of a decade ago, "you could fillet these **fish** again, and that made people very, very happy," Mr. Paulson says.

Saturday morning I arise at 8:30 and click on "America's best-known **fish**erman," giving advice on catching **bass** in cold weather from the seat of a bass boat in Louisiana.

Figure 17.2 Samples of *bass* sentences extracted from the WSJ using the simple correlates *play* and *fish*.

presents results that show it yields remarkably good results. As an illustration of this technique, consider the situation where we would like to generate a reasonable set of seed sentences for the fish and musical senses of *bass*. Without too much thought, we might come up with *fish* as a reasonable indicator of **bass**^{1}, and *play* as a reasonable indicator of **bass**^{2}. Figure 17.2 shows a partial result of a such a search for the strings "fish" and "play" in a corpus of *bass* examples drawn from the WSJ.

Of course, we might also want some way to automatically suggest these associated words. Yarowsky (1995) suggests two methods to select effective correlates: deriving them from machine readable dictionary entries, and selecting seeds using collocational statistics such as those described in Chapter 6. Yarowsky (1995) reports an average performance of 96.5% on a

coarse binary sense assignment involving 12 words. In these experiments, a training set derived using bootstrapping with seed sentences discovered using correlates was used to train a decision list classifier for each word.

Unsupervised Methods: Discovering Word Senses

Unsupervised approaches to sense disambiguation eschew the use of sense tagged data of any kind during training. In these approaches, feature-vector representations of unlabeled instances are taken as input and are then grouped into clusters according to a similarity metric. These clusters can then be represented as the average of their constituent feature-vectors, and labeled by hand with known word senses. Unseen feature-encoded instances can be classified by assigning them the word sense from the cluster to which they are closest according to the similarity metric.

Fortunately, clustering is a well-studied problem with a wide number of standard algorithms that can be applied to inputs structured as vectors of numerical values (Duda and Hart, 1973). A frequently used technique in language applications is known as **agglomerative clustering**. In this technique, each of the N training instances is initially assigned to its own cluster. New clusters are then formed in a bottom-up fashion by successively merging the two clusters that are most similar. This process continues until either a specified number of clusters is reached, or some global goodness measure among the clusters is achieved. In cases where the number of training instances makes this method too expensive, random sampling can be used on the original training set (Cutting et al., 1992b) to achieve similar results.

AGGLOMERATIVE CLUSTERING

The fact that these unsupervised methods do not make use of hand-labeled data poses a number of challenges for evaluating any clustering result. The following problems are among the most important ones that have to be addressed in unsupervised approaches:

- The correct senses of the instances used in the training data may not be known.
- The clusters are almost certainly heterogeneous with respect to the senses of the training instances contained within them.
- The number of clusters is almost always different from the number of senses of the target word being disambiguated.

Schütze's experiments (Schütze, 1992, 1998) constitute an extensive application of unsupervised clustering to word sense disambiguation. Although the actual technique is quite involved, unsupervised clustering is at the core of the method. Schütze's results indicate that for coarse binary dis-

tinctions, unsupervised techniques can achieve results approaching those of supervised and bootstrap methods, in most instances approaching the 90% range. As with most of the supervised methods, this method was tested on a small sample of words.

Dictionary-Based Approaches

A major drawback with all of these approaches is the problem of scale. All require a considerable amount of work to create a classifier for each ambiguous entry in the lexicon. For this reason, most of the experiments with these methods report results ranging from 2 to 12 lexical items (The work of Ng and Lee (1996) is a notable exception reporting results disambiguating 121 nouns and 70 verbs). Scaling up any of these approaches to deal with all the ambiguous words in a language would be a large undertaking. Instead, attempts to perform large-scale disambiguation have focused on the use of **machine readable dictionaries**, of the kind discussed in Chapter 16. In this style of approach, the dictionary provides both the means for constructing a sense tagger, and the target senses to be used.

The first implementation of this approach is due to Lesk (1986). In this approach, all the sense definitions of the word to be disambiguated are retrieved from the dictionary. Each of these senses is then compared to the dictionary definitions of all the remaining words in the context. The sense with the highest overlap with these context words is chosen as the correct sense. Note that the various sense definitions of the context words are all simply lumped together in this approach.

To make this more concrete, consider Lesk's example of selecting the appropriate sense of *cone* in the phrase *pine cone* given the following definitions for *pine* and *cone*.

pine 1 kinds of evergreen tree with needle-shaped leaves
2 waste away through sorrow or illness
cone 1 solid body which narrows to a point
2 something of this shape whether solid or hollow
3 fruit of certain evergreen trees

In this example, Lesk's method would select **cone**[3] as the correct sense since two of the words in its entry, *evergreen* and *tree*, overlap with words in the entry for *pine*, whereas neither of the other entries have any overlap with words in the definition of *pine*. Lesk reports accuracies of 50-70% on short samples of text selected from Austen's *Pride and Prejudice* and an AP newswire article.

The primary problem with this approach is that the dictionary entries for the target words are relatively short, and may not provide sufficient material to create adequate classifiers since the words used in the context and their definitions must have direct overlap with the words contained in the appropriate sense definition in order to be useful.[1] One way to remedy this problem is to expand the list of words used in the classifier to include words related to, but not contained in their individual sense definitions. This can be accomplished by including words whose definitions make use of the target word. For example, the word *deposit* does not occur in the definition of *bank* in the American Heritage Dictionary (Morris, 1985). However, *bank* does occur in the definition of *deposit*. Therefore, the classifier for *bank* can be expanded to include *deposit* as a relevant feature.

Of course, just knowing that *deposit* is related to *bank* does not help much since we don't know to which sense of *bank* it is related. Specifically, to make use of *deposit* as a feature, we have to know which sense of *bank* was being used in its definition. Fortunately, many dictionaries and thesauri include tags known as **subject codes** in their entries that correspond roughly to broad conceptual categories. For example, the entry for *bank* in the *Longman's Dictionary of Contemporary English* (LDOCE) (Procter, 1978) includes the subject code EC (Economics) for the financial senses of *bank*. Given such subject codes, we can guess that expanded terms with the subject code EC will be related to this sense of bank rather than any of the others. Guthrie et al. (1991) report results ranging from 47% correct for fine-grained LDOCE distinctions to 72% for more coarse distinctions.

SUBJECT CODES

17.3 INFORMATION RETRIEVAL

Information retrieval is a growing field that encompasses a wide range of topics related to the storage and retrieval of all manner of media. The focus of this section is with the storage of text documents and their subsequent retrieval in response to users' requests for information. Of particular interest is the widespread adoption of word-based indexing and retrieval methods. Most current information retrieval systems are based on an extreme interpretation of the principle of compositional semantics. In these systems, the meaning of documents resides solely in the words that are contained within

[1] Indeed, Lesk (1986) notes that the performance of his system seems to roughly correlate with the length of the dictionary entries.

them. To revisit the Mad Hatter's quote from the beginning of Chapter 16, in these systems *I see what I eat* and *I eat what I see* mean precisely the same thing. The ordering and constituency of the words that make up the sentences that make up documents play no role in determining their meaning. Because they ignore syntactic information, these approaches are often referred to as **bag of words** methods.

BAG OF WORDS

Before moving on, we need to introduce some new terminology. In information retrieval, a **document** refers generically to the unit of text indexed in the system and available for retrieval. Depending on the application, a document can refer to anything from intuitive notions like newspaper articles, or encyclopedia entries, to smaller units such as paragraphs and sentences. In Web-based applications, it can refer to a Web page, a part of a page, or to an entire Website. A **collection** refers to a set of documents being used to satisfy user requests. A **term** refers to a lexical item that occurs in a collection, but it may also include phrases. Finally, a **query** represents a user's information need expressed as a set of terms.

DOCUMENT

COLLECTION

TERM

QUERY

The specific information retrieval task that we will consider in detail is known as **ad hoc retrieval**. In this task, it is assumed that an unaided user poses a query to a retrieval system, which then returns a possibly ordered set of potentially useful documents. Several other related, lexically oriented, information retrieval tasks will be discussed in Section 17.4.

AD HOC RETRIEVAL

The Vector Space Model

In the **vector space model** of information retrieval, documents and queries are represented as vectors of features representing the terms that occur within them (Salton, 1971). More properly, they are represented as vectors of features consisting of the terms that occur *within the collection*, with the value of each feature indicating the presence or absence of a given term in a given document. These vectors can be represented as follows:

VECTOR SPACE MODEL

$$\vec{d}_j = (t_{1,j}, t_{2,j}, t_{3,j}, \cdots, t_{N,j})$$
$$\vec{q}_k = (t_{1,k}, t_{2,k}, t_{3,k}, \cdots, t_{N,k})$$

In this notation, $\vec{d}_j$ and $\vec{q}_k$ denote a particular document and query, while the various t features represent the N terms that occur in the collection as a whole. Let's first consider the case where these features take on the value of one or zero, indicating the presence or absence of a term in a document or query. Given this approach, a simple way to determine the relevance of a document to a query is to determine the number of terms they have in

common. This can be accomplished by the following similarity metric:

$$sim(\vec{q}_k, \vec{d}_j) = \sum_{i=1}^{N} t_{i,k} \times t_{i,j} \tag{17.17}$$

In this equation, the similarity between the query vector, $\vec{q}_k$ and the document vector, $\vec{d}_j$, is measured by simply summing the number of terms they share.

Of course, a problem with the use of binary values for features is that it fails to capture the fact that some terms are more important to the meaning of a document than others. A useful generalization is to replace the ones and zeroes with numerical **weights** that indicate the importance of the various terms in particular documents and queries. We can thus generalize our vectors as follows:

WEIGHTS

$$\vec{d}_j = (w_{1,j}, w_{2,j}, w_{3,j}, \cdots, w_{n,j})$$

$$\vec{q}_k = (w_{1,k}, w_{2,k}, w_{3,k}, \cdots, w_{n,k})$$

This characterization of documents as vectors of term weights allows us to view the document collection as a whole as a matrix of weights, where $w_{i,j}$ represents the weight of term i in document j. This weight matrix is typically called a **term-by-document matrix** matrix. Under this view, the columns of the matrix represent the documents in the collection, and the rows represent the terms.

TERM-BY-DOCUMENT MATRIX

It is useful to view the features used to represent documents (and queries) in this model as dimensions in a multi-dimensional space. Correspondingly, the weights that serve as values for those features serve to locate documents in that space. When a user's query is translated into a vector it denotes a point in that space. Documents that are located close to the query can then be judged as being more relevant than documents that are farther away.

This characterization of documents and queries as vectors provides all the basic parts for an ad hoc retrieval system. A document retrieval system can simply accept a user's query, create a vector representation for it, compare it against the vectors representing all known documents, and sort the results. The result is a list of documents rank ordered by their similarity to the query.

Consider as an example of this approach, the space shown in Figure 17.3. This figure shows a simplified space consisting of the three dimensions corresponding to the terms *speech*, *language* and *processing*. The three vectors illustrated in this space represent documents derived from the chapter and section headings of Chapters 1, 7, and 13 of this text, which we will denote as **Doc1**, **Doc7**, and **Doc13**, respectively. If we use raw term frequency

in document as a weight, then **Doc1** is represented by the vector $(1,2,1)$, **Doc7** by $(6,0,1)$, and **Doc13** by $(0,5,1)$. As is clear from the figure, this space captures certain intuitions about how these chapters are related. Chapter 1, being general, is fairly similar to both Chapters 7 and 13. Chapters 7 and 13, on the other hand, are distant from one another since they cover a different set of topics.

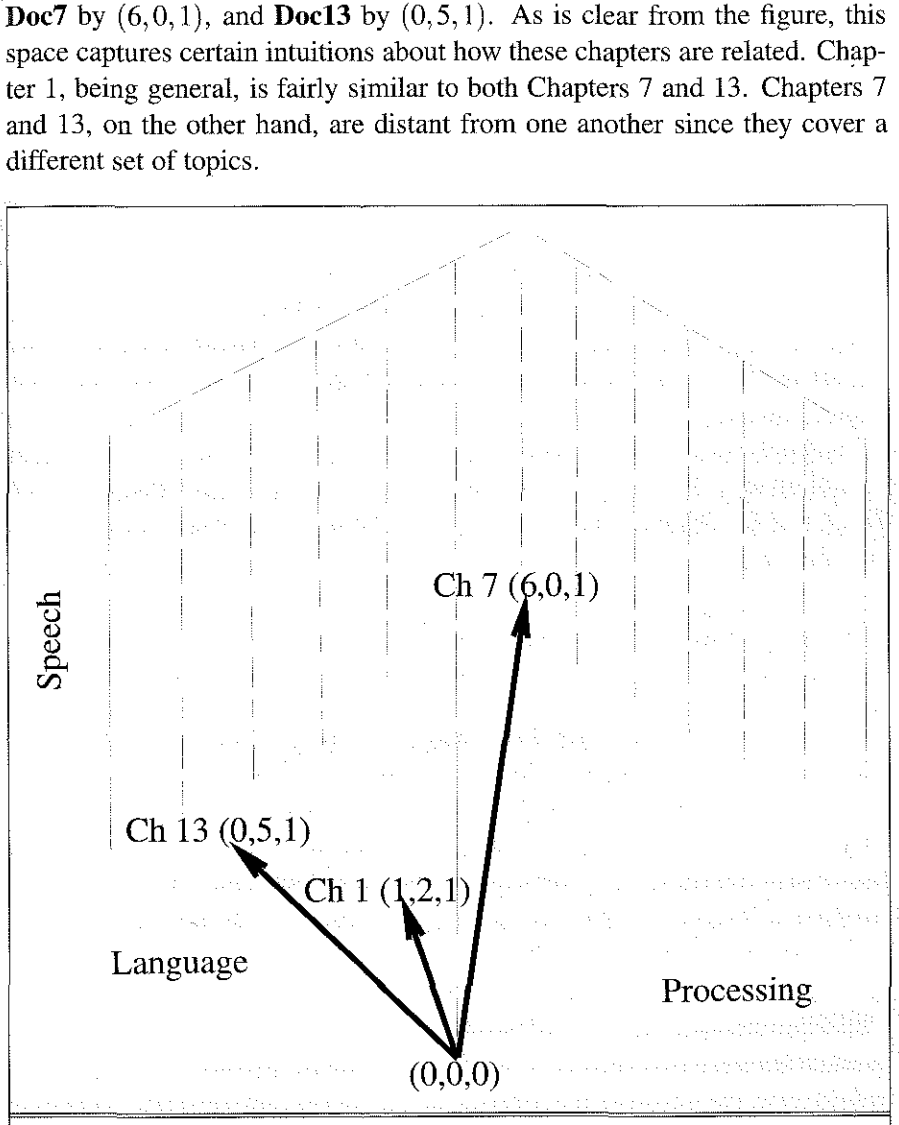


Figure 17.3 A simple vector space representation of documents derived from the text of the chapter and section headings of Chapters 1, 7, and 13 in three dimensions.

Unfortunately, this instantiation of a vector space places too much emphasis on the absolute values of the various coordinates of each document. For example, what is important about the *speech* dimension of the **Doc7**, is

not the value 6 but rather that it is the dominant contributor to the meaning of that document. Similarly, the specific values of 1, 2, and 1 for **Doc1** are not important, what is important is that the three dimensions have roughly similar weights. It would be sensible, for example, to assume that a new document with weights 3, 6, and 3 would be quite similar to **Doc1** despite the magnitude differences in the term weights.

We can accomplish this effect by **normalizing** document vectors. By normalizing, we simply mean converting all the vectors to a standard length. Converting to a unit length can be accomplished by dividing each of their dimensions by the overall length of the vector, which is defined as $\sum_{i=1}^{N} w_i^2$. This, in effect, eliminates the importance of the exact length of a document's vector in the space, and emphasizes instead the direction of the document vector with respect to the origin.

Applying this technique to our three sample documents results in the following term-by-document matrix, A, where the columns represent **Doc1**, **Doc7** and **Doc13** and the rows represent the terms *speech*, *language*, and *processing*:

$$A = \begin{pmatrix} .41 & .81 & .41 \\ .98 & 0 & .16 \\ 0 & .98 & .19 \end{pmatrix}$$

You should verify that with this scheme, the normalized vectors for **Doc1** and our hypothetical $(3,6,3)$ document end up as identical vectors.

Now let's return to the topic of determining the similarity between vectors. Updating the similarity metric given earlier with numerical weights rather than binary values, gives us the following equation:

$$sim(\vec{q}_k, \vec{d}_j) = \vec{q}_k \cdot \vec{d}_j = \sum_{i=1}^{N} w_{i,k} \times w_{i,j} \tag{17.18}$$

DOT PRODUCT

This equation specifies the **dot product** between vectors. In general, the dot product between two vectors is of no use as a similarity metric, since it is too sensitive to the absolute magnitudes of the various dimensions. However, the dot product between vectors that have been normalized has a useful and intuitive interpretation: it computes the **cosine** of the angle between two vectors. When two documents are identical they will receive a cosine of one; when they are orthogonal (share no common terms) they will receive a cosine of zero.

COSINE

Note that if for some reason the vectors are not stored in a normalized form, then the normalization can be incorporated directly into the similarity

measure as follows:

$$sim(\vec{q}_k, \vec{d}_j) = \frac{\sum_{i=1}^{N} w_{i,k} \times w_{i,j}}{\sqrt{\sum_{i=1}^{N} w_{i,k}^2} \times \sqrt{\sum_{i=1}^{N} w_{i,j}^2}} \tag{17.19}$$

Of course, in situations where the document collection is relatively static and many queries are being performed, it makes sense to normalize the document vectors once and store them, rather than include the normalization in the similarity metric.

Let's consider how this similarity metric would work in the context of some small examples. Consider the carefully selected query consisting solely of the terms *speech*, *language* and *processing*. Converting this query to a vector and normalizing it results in the vector $(.57, .57, .57)$. Computing the cosines between this vector and our three document vectors shows that **Doc1** is closest with a cosine of .92, followed by **Doc13** with a cosine of .67, and finally **Doc7** with a cosine of .65. Not surprisingly, this ranking is in close accord with our intuitions about the relationship between this query and these documents.

Now consider a shorter query consisting solely of the terms *speech* and *processing*. Processing this query yields the normalized vector $(.70, 0, .70)$. When the cosines are computed between this vector and our documents, **Doc7** is now the closest with a cosine of .80, followed by **Doc1** with a score of .58, with **Doc13** coming in a distant third with a cosine of .13.

Term Weighting

In practice, the method used to assign terms weights in the document and query vectors has an enormous impact on the effectiveness of a retrieval system. Two factors have proven to be critical in deriving effective term weights: term frequency within a single document, and the distribution of terms across a collection. We can begin with the simple notion that terms which occur frequently within a document may reflect its meaning more strongly than terms that occur less frequently and should thus have higher weights. In its simplest form, this factor is called **term frequency** and is simply the raw frequency of a term within a document (Luhn, 1957).

TERM FREQUENCY

The second factor to consider is the distribution of terms across the collection as a whole. Terms that are limited to a few documents are useful for discriminating those documents from the rest of the collection. On the other hand, terms that occur frequently across the entire collection are less useful in discriminating among documents. What is needed therefore is a

METHODOLOGY BOX: EVALUATING INFORMATION RETRIEVAL SYSTEMS

Information retrieval systems are evaluated with respect to the notion of **relevance** — *a judgment by a human* that a document is relevant to a query. A system's ability to retrieve relevant documents is assessed with a **recall** measure, as in Chapter 15.

$$\textbf{Recall} = \frac{\text{\# of relevant documents returned}}{\text{total \# of relevant documents in the collection}}$$

Of course, a system can achieve 100% recall by simply returning all the documents in the collection. A system's accuracy is based on how many of the documents returned for a given query are actually relevant, which can be assessed by a **precision** metric.

$$\textbf{Precision} = \frac{\text{\# of relevant documents returned}}{\text{\# of documents returned}}$$

These measures are complicated by the fact that most systems do not make explicit relevance judgments, but rather rank their collection with respect to a query. To deal with this we can specify a set of cutoffs in the output, and measure average precision for the documents ranked above the cutoff. Alternatively, we can specify a set of recall levels and measure average precision at those levels. This latter method gives rise to what are known as precision-recall curves as shown in Figure 17.4. As these curves show, comparing the performance of two systems can be difficult. In this comparison, one system is better at both high and low levels of recall, while the other is better in the middle region. An alternative to these curves are metrics that attempt to combine recall and precision into a single value. The F measure introduced on page 578 is one such measure.

The U.S. government-sponsored TREC (Text REtrieval Conference) evaluations have provided a rigorous testbed for the evaluation of a variety of information retrieval tasks and techniques. Like the MUC evaluations, TREC provides large document sets for both training and testing, along with a uniform scoring system. Training materials consist of sets of documents accompanied by sets of queries (called topics in TREC) and relevance judgments. Voorhees and Harman (1998) provide the details for the most recent meeting. Details of all of the meetings can be found at the TREC page on the National Institute of Standards and Technology Website.

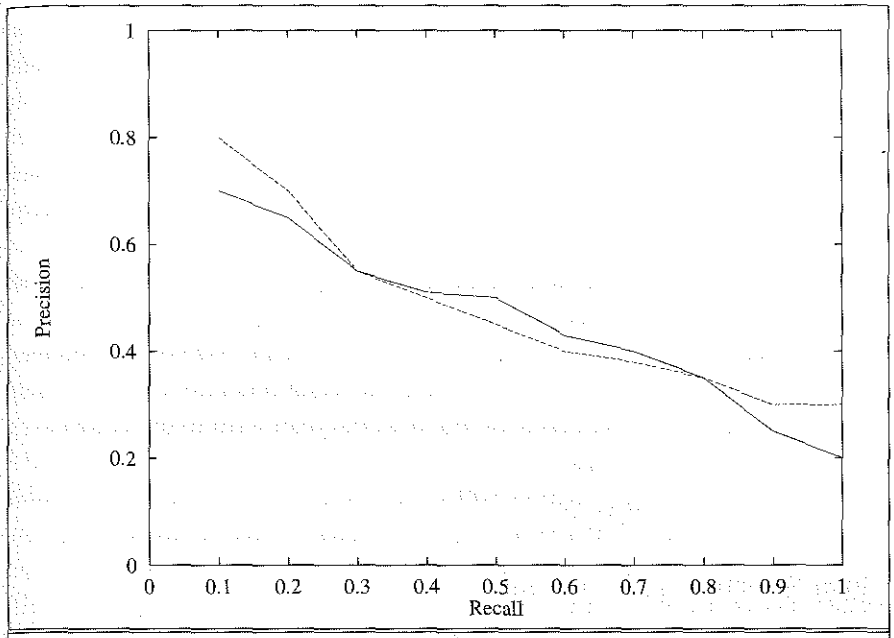


Figure 17.4 Precision-recall curves for two hypothetical systems. These curves plot the average precision of a set of returned documents at a given level of recall. For example, with both of these systems, drawing a cutoff in the return set at the document where they achieve 30% recall results in an average precision of 55%.

measure that favors terms which occur in fewer documents. The fraction N/n_i, where N is the total number of documents in the collection, and n_i is the number of documents in which term i occurs, provides exactly this measure. The fewer documents a term occurs in, the higher this weight. The lowest weight of 1 is assigned to terms that occur in all the documents. Due to the large number of documents in many collections, this measure is usually squashed with a log function leaving us with the following **inverse document frequency** term weight (Sparck Jones, 1972):

INVERSE DOCUMENT FREQUENCY

$$\text{idf}_i = \log\left(\frac{N}{n_i}\right) \tag{17.20}$$

Combining the term frequency factor with this factor results in a scheme known as tf · idf weighting:

$$w_{i,j} = \text{tf}_{i,j} \times \text{idf}_i \tag{17.21}$$

That is, the weight of term i in the vector for document j is the product of its overall frequency in j with the log of its inverse document frequency in

the collection. With some minor variations, this weighting scheme is used to assign term weights to documents in nearly all vector space retrieval models.

Despite the fact that we use the same representations for documents and queries, it is not at all clear that the same weighting scheme should be used for both. In many ad hoc retrieval settings, such as Web search engines, user queries are not very much like documents at all. For example, an analysis of a very large set of queries (1,000,000,000 actually) from the AltaVista search engine reveals that the average query length is around 2.3 words (Silverstein et al., 1998). In such an environment, the raw term frequency in the query is not likely to be a very useful factor. Instead, Salton and Buckley (1988) recommend the following formula for weighting query terms, where $\max_j \mathrm{tf}_{j,k}$ denotes the frequency of the most frequent term in document k.

$$w_{i,k} = \left(0.5 + \frac{0.5\mathrm{tf}_{i,k}}{\max_j \mathrm{tf}_{j,k}}\right) \times \mathrm{idf}_i \tag{17.22}$$

Term Selection and Creation

Thus far, we have been assuming that it is precisely the words that occur in a collection that are used to index the documents in the collection. Two common variations on this assumption involve the use of **stemming**, and a **stop list**.

STEMMING

The notion of **stemming** takes us back to Chapter 3 and the topic of morphological analysis. The basic question addressed by stemming is whether the morphological variants of a lexical item should be listed (and counted) separately, or whether they should be collapsed into a single root form. For example, without stemming, the terms *process*, *processing* and *processed* will be treated as distinct items with separate term frequencies in a term-by-document matrix; with stemming they will be conflated to the single term *process* with a single summed frequency count. The major advantage to using stemming is that it allows a particular query term to match documents containing any of the morphological variants of the term. The Porter stemmer (Porter, 1980) described in Chapter 3 is frequently used for retrieval from collections of English documents.

A problem with this approach is that it throws away useful distinctions. For example, consider the use of the Porter stemmer on documents and queries containing the words *stocks* and *stockings*. In this case, the Porter stemmer reduces these surface forms to the single term *stock*. Of course, the result of this is that queries concerning *stock prices* will return documents about *stockings*, and queries about *stockings* will find documents

about *stocks*.[2] More technically, stemming may increase recall by finding documents with terms that are morphologically related to queries, but it may also reduce precision by returning semantically unrelated documents. For this reason, few Web search engines currently make use of stemming. Hull (1996) presents results from a series of experiments that explore the efficacy of stemming.

A second common technique involves the use of stop lists, which address the issue of what words should be allowed into the index. A **stop list** STOP LIST is simply a list of high frequency words that are eliminated from the representation of both documents and queries. Two motivations are normally given for this strategy: high frequency, closed-class terms are seen as carrying little semantic weight and are thus unlikely to help with retrieval, and eliminating them can save considerable space in the inverted index files used to map from terms to the documents that contain them. The downside of using a stop list is that it makes it difficult to search for phrases that contain words in the stop list. For example, a common stop list derived from the Brown corpus presented in Frakes and Baeza-Yates (1992), would reduce the phrase *to be or not to be* to the phrase *not*.

Homonymy, Polysemy, and Synonymy

Since the vector space model is based solely on the use of simple terms, it is useful to consider the effect that various lexical semantic phenomena may have on the model. Consider a query containing the word *canine* with its *tooth* and *dog* senses. A query containing *canine* will be judged similar to documents making use of either of these senses. However, given that users are probably only interested in one of these senses, the documents containing the other sense will be judged non-relevant. Homonymy and polysemy, therefore, can have the effect of *reducing precision* by leading a system to return documents irrelevant to the user's information need.

Now consider a query consisting of the lexeme *dog*. This query will be judged close to documents that make frequent use of the term *dog*, but may fail to match documents that use close synonyms like *canine*, as well as documents that use hyponyms such as *Malamute*. Synonymy and hyponymy, therefore, can have the effect of *reducing recall* by causing the retrieval sys-

[2] This example is motivated by some bad publicity received by a well-known search engine, when it returned some rather salacious sites containing extensive use of the term *stockings* in response to queries concerning *stock prices*. In response, a spokesman announced that their engineers were working hard on a solution to this strange problem with words.

tem to miss relevant documents.

Note that it is inaccurate to state flatly that polysemy reduces precision, and synonymy reduces recall since, as we discussed on page 652, both measures are relative to a fixed cutoff. As a result, every non-relevant document that rises above the cutoff due to polysemy takes up a slot in the fixed size return set, and may thus push a relevant document below threshold, thus reducing recall. Similarly, when a document is missed due to synonymy, a slot is opened in the return set for a non-relevant document, potentially reducing precision as well.

These issues lead naturally to the question of whether or not word sense disambiguation can help in information retrieval. The current evidence on this point is mixed, with some experiments reporting a gain using disambiguation-like techniques (Schütze and Pedersen, 1995), and others reporting either no gain, or a degradation in performance (Krovetz and Croft, 1992; Sanderson, 1994; Voorhees, 1998).

Improving User Queries

One of the most effective ways to improve retrieval performance is to find a way to improve user queries. The techniques presented in this section have been shown to varying degrees to be effective at this task.

RELEVANCE FEEDBACK

The single most effective way to improve retrieval performance in the vector space model is the use of **relevance feedback** (Rocchio, 1971). In this method, a user presents a query to the system and is presented with a small set of retrieved documents. The user is then asked to specify which of these documents appears relevant to their need. The user's original query is then reformulated based on the distribution of terms in the relevant and non-relevant documents that the user examined. This reformulated query is then passed to the system as a *new* query with the new results being shown to the user. Typically an enormous improvement is seen after a single iteration of this technique.

The formal basis for the implementation of this technique falls out directly from some of the basic geometric intuitions of the vector model. In particular, we would like to *push* the vector representing the user's original query toward the documents that have been found to be relevant, and away from the documents judged not relevant. This can be accomplished by adding an averaged vector representing the relevant documents to the original query, and subtracting an averaged vector representing the non-relevant documents.

More formally, let's assume that $\vec{q}_i$ represents the user's original query, R is the number of relevant documents returned from the original query, S is the number of non-relevant documents, and documents in the relevant and non-relevant sets are denoted as $\vec{r}$ and $\vec{s}$, respectively. In addition, assume that β and γ range from 0 to 1 and that $\beta + \gamma = 1$. Given these assumptions, the following represents a standard relevance feedback update formula:

$$\vec{q}_{i+1} = \vec{q}_i + \frac{\beta}{R}\sum_{j=1}^{R}\vec{r}_j - \frac{\gamma}{S}\sum_{k=1}^{S}\vec{s}_k$$

The factors β and γ in this formula represent parameters that can be adjusted experimentally. Intuitively, β represents how far the new vector should be pushed towards the relevant documents, and γ represents how far it should be pushed away from the non-relevant ones. Salton and Buckley (1990) report good results with $\beta = .75$ and $\gamma = .25$.

We should note that evaluating systems that use relevance feedback is rather tricky. In particular, an enormous improvement is often seen in the documents retrieved by the first reformulated query. This should not be too surprising since it includes the documents that the user told the system were relevant on the first round. The preferred way to avoid this inflation is to only compute recall and precision measures for what is called the **residual collection**, the original collection without any of the documents shown to the user on any previous round. This usually has the effect of driving the system's raw performance below that achieved with the first query, since the most highly relevant documents have now been eliminated. Nevertheless, this is an effective technique to use when comparing distinct relevance feedback mechanisms.

RESIDUAL COLLECTION

An alternative approach to query improvement focuses on the terms that comprise the query vector, rather than the query vector itself. In **query expansion**, the user's original query is expanded to include terms related to the original terms. This has typically been accomplished by adding terms chosen from lists of terms that are highly correlated with the user's original terms in the collection. Such highly correlated terms are listed in what is typically called a **thesaurus**, although since it is based on correlation, rather than synonymy, it is only loosely connected to the standard references that carry the same name.

QUERY EXPANSION

THESAURUS

Unfortunately, it is usually the case that available thesaurus-like resources are not suitable for most collections. In **thesaurus generation**, a correlation-based thesaurus is generated automatically from all or a portion of the documents in the collection. Not surprisingly, one of the most popular

THESAURUS GENERATION

TERM CLUSTERING

methods used in thesaurus generation involves the use of **term clustering**. Recall from our characterization of the term-by-document matrix that the columns in the matrix represent the documents and the rows represent the terms. Therefore, in thesaurus generation, the rows can be clustered to form sets of synonyms, which can then be added to the user's original query to improve its recall.

This technique is typically instantiated in one of two ways: a thesaurus can be generated once from the document collection as a whole (Crouch and Yang, 1992), or sets of synonym-like terms can be generated dynamically from the returned set for the original query (Attar and Fraenkel, 1977). Note that this second approach entails far more effort, since in effect a small thesaurus is generated for the documents returned for every query, rather than once for the entire collection.

17.4 Other Information Retrieval Tasks

As noted earlier, ad-hoc retrieval is not the only word-based task in information retrieval. Some of the other more important ones include document categorization, document clustering, text segmentation, and summarization.

DOCUMENT CATEGORIZATION

The **document categorization** task is to assign a new document to one of a pre-existing set of document classes. In this setting, the task of creating a classifier consists of discovering a useful characterization of the documents that belong in each class. Although this can be done by hand, the standard approach is to use supervised machine learning. In particular, classifiers can be trained on a set of documents that have been labeled with the correct class. Any of the supervised learning methods introduced on page 638 for word sense disambiguation can be applied to this task as well. When categorization is performed with the intent of transmitting a document to a user, or set of interested users, it is usually referred to as **routing**. The term **filtering** is used in the special case where the categorization task is to either accept or reject a document, as in e-mail filters that attempt to screen for junk mail.

ROUTING

FILTERING

DOCUMENT CLUSTERING

The categorization task assumes an existing classification, or clustering, of documents. By contrast, the task of **document clustering** is to create, or discover, a reasonable set of clusters for a given set of documents. As was the case in word sense discovery, a reasonable cluster is defined as one that maximizes the within-cluster document similarity, and minimizes between-cluster similarity. There are two principal motivations for the use of this

technique in an ad hoc retrieval setting: efficiency, and the **cluster hypothesis**.

The efficiency motivation arises from the enormous size of many modern document collections. Recall that the retrieval method described in the last section requires every query to be compared against every document in the collection. If a collection can be divided up into a set of N conceptually coherent clusters, then queries could first be compared against representations of each of the N clusters. Ordinary retrieval could then be applied only within the top cluster or clusters, thus saving the cost of comparing the query to the documents in all of the other more distant clusters.

CLUSTER HYPOTHESIS

The **cluster hypothesis** (Jardine and van Rijsbergen, 1971) takes this argument a step further by asserting that retrieval from a clustered collection will not only be more efficient, but will in fact improve retrieval performance in terms of recall and precision. The basic notion behind this hypothesis is that by separating documents according to topic, relevant documents will be found together in the same cluster, and non-relevant documents will be avoided since they will reside in clusters that are not used for retrieval. Despite the plausibility of this hypothesis, there is only mixed empirical support for it. Results vary considerably based on the clustering algorithm and document collection in use (Willett, 1988; Shaw et al., 1996).

A promising alternative application of clustering is to cluster the documents returned in response to a user's query, rather than the document collection as whole. Hearst and Pedersen (1996) present evidence that this technique provides many of benefits promised by the cluster hypothesis.

TEXT SEGMENTATION

In **text segmentation**, larger documents are automatically broken down into smaller semantically coherent chunks. This is useful in domains where there are a significant number of large documents that cover a wide variety of topics. Text segmentation can be used to either perform retrieval below the document level, or to visually guide the user to relevant parts of retrieved documents. Again, not surprisingly, segmentation algorithms often make use of vector-like representations for the subparts of a larger document. Adjacent subparts that have similar cosines are more likely to be about the same topic than adjacent segments with more distant cosines. Roughly speaking, such discontinuities in the similarity between adjacent text segments can be used to divide larger documents into subparts (Salton et al., 1993; Hearst, 1997).

TEXT SUMMARIZATION

Finally, the task of **text summarization** (Sparck Jones, 1997) is to produce a shorter, summary version of an original document. In general, two approaches have been taken to this problem. In the **knowledge-based**

approach, the original document undergoes a semantic analysis which produces a representation of the meaning of the text. This representation is then passed to a text generator which produces a summary text that conveys the important points of the original and satisfies given length restrictions. More details on text generation are presented in Chapter 20. In **selection-based summarization**, a summary document is created by first assigning a importance weight to all the sentences from the original document according to very simple word frequency and discourse structure heuristics. A summary document is then generated by determining a threshold such that the inclusion of all sentences above the threshold results in a document with the desired size.

SELECTION-BASED SUMMARIZATION

17.5 SUMMARY

This chapter has explored two major areas of lexical semantic processing: word sense disambiguation and information retrieval.

- **Word sense disambiguation** systems assign word tokens in context to one of a pre-specified set of senses.
- **Selectional restriction-based** approaches can be used to disambiguate both predicates and arguments, but require considerable information about semantic roles restrictions and hierarchical type information about role fillers.
- **Machine learning** approaches to sense disambiguation make it possible to automatically create robust sense disambiguation systems.
  - **Supervised** approaches use collections of texts annotated with their correct senses to train classifiers.
  - **Bootstrapping** approaches permit the use of supervised methods with far fewer resources.
  - **Unsupervised** clustering-based approaches attempt to discover representations of word senses from unannotated texts.
- **Machine readable dictionaries** facilitate the creation of broad-coverage sense disambiguators.
- The dominant models of information retrieval represent the meanings of documents and queries as bags of words.
- The **vector space model** views documents and queries as vectors in a large multidimensional space. In this model, the similarity between

documents and queries, or other documents, can be measured by the cosine of the angle between the vectors.

- User queries can be improved through query reformulation using either **relevance feedback** or thesaurus-based query expansion.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Word sense disambiguation traces its roots to some of the earliest applications of digital computers. The notion of disambiguating a word by looking at a small window around it was apparently first suggested by Warren Weaver (1955), in the context of machine translation. Among the notions first proposed in this early period were the use of a thesaurus for disambiguation (Masterman, 1957), supervised training of Bayesian models for disambiguation (Madhu and Lytel, 1965), and the use of clustering in word sense analysis (Sparck Jones, 1986).

An enormous amount of work on disambiguation has been conducted within the context of AI-oriented natural language processing systems. Most natural language analysis systems of this type exhibit some form of lexical disambiguation capability, however, a number of these efforts made word sense disambiguation a larger focus of their work. Among the most influential efforts were the efforts of Quillian (1968) and Simmons (1973) with semantic networks, the work of Wilks with *Preference Semantics* Wilks (1975c, 1975b, 1975a), and the work of Small and Rieger (1982) and Riesbeck (1975) on word-based understanding systems. Hirst's ABSITY system (Hirst and Charniak, 1982; Hirst, 1987, 1988), which used a technique based on semantic networks called marker passing, represents the most advanced system of this type. As with these largely symbolic approaches, most connectionist approaches to word sense disambiguation have relied on small lexicons with hand-coded representations (Cottrell, 1985; Kawamoto, 1988).

We should note that considerable work on sense disambiguation has been conducted in the areas of Cognitive Science and psycholinguistics. Appropriately enough, it is generally described using a different name: lexical ambiguity resolution. Small et al. (1988) present a variety of papers from this perspective.

The earliest implementation of a robust empirical approach to sense disambiguation is due to Kelly and Stone (1975) who directed a team that hand-crafted a set of disambiguation rules for 1790 ambiguous English words.

Lesk (1986) was the first to use a machine readable dictionary for word sense disambiguation. The efforts at New Mexico State University using LDOCE are among the most extensive explorations of the use of machine readable dictionaries. Much of this work is described in Wilks et al. (1996). The problem of dictionary senses being too fine-grained or lacking an appropriate organization has been addressed in the work of Dolan (1994) and Chen and Chang (1998).

Modern interest in supervised machine learning approaches to disambiguation began with Black (1988), who applied decision tree learning to the task. The need for large amounts of annotated text in these methods led to investigations into the use of bootstrapping methods (Hearst, 1991; Yarowsky, 1995). The problem of how to weight and combine the disparate sources of evidence used in many robust systems is explored in Ng and Lee (1996) and McRoy (1992). There has been considerably less work in the area of unsupervised methods. The earliest attempt to use clustering in the study of word senses is due to Sparck Jones (1986). Zernik (1991) successfully applied a standard information retrieval clustering algorithm to the problem, and provided an evaluation based on improvements in retrieval performance. More extensive recent work on clustering can be found in Pedersen and Bruce (1997) and Schütze (1997, 1998).

Note that of all of these robust efforts, only three have attempted to exploit the power of mutually disambiguating all the words in a sentence. The system described in Kelly and Stone (1975) makes multiple passes over a sentence to take later advantage of easily disambiguated words; Cowie et al. (1992) use a simulated annealing model to perform a parallel search for a desirable set of senses; Veronis and Ide (1990) use inhibition and excitation in a neural network automatically constructed from a machine readable dictionary.

Ide and Veronis (1998) provide a comprehensive review of the history and current state of word sense disambiguation. Ng and Zelle (1997) provide a more focused review from a machine learning perspective. Wilks et al. (1996) describe a wide array of dictionary and corpus-based experiments, along with detailed descriptions of some very early work.

Luhn (1957) is generally credited with first advancing the notion of fully automatic indexing of documents based on their contents. Over the years Salton's SMART project (Salton, 1971) at Cornell developed or evaluated many of the most important notions in information retrieval including the vector model, term weighting schemes, relevance feedback, and the use of cosine as a similarity metric. The notion of using inverse document fre-

quency in term weighting is due to Sparck Jones (1972). The original notion of relevance feedback is due to Rocchio (1971). An alternative to the vector model that we have not covered is the **probabilistic model**. Originally shown effective by Robinson and Sparck Jones (1976), a Bayesian network version of the probabilistic model is the basis for the widely used INQUERY system (Callan et al., 1992). Crestani et al. (1998) present a comprehensive review of probabilistic models in information retrieval.

PROBABILISTIC MODEL

The cluster hypothesis was introduced in Jardine and van Rijsbergen (1971). Willett (1988) provides a critical review of the major efforts in this area. Mather (1998) presents an algorithm-independent clustering metric that can be used to evaluate the performance of various clustering algorithms. A collection of papers on document categorization and its close siblings, filtering and routing, can be found in Lewis and Hayes (1994). A recent example of routing is AT&T's "How May I Help You?" task where the goal is to classify a user's utterance into one of fifteen possible categories, such as third number billing, or collect call. Once the system has classified the call, the system routes the caller to an appropriate human operator. The classification accuracy on this task approaches 80%, despite the fact that the speech recognizer has a word accuracy rate of only around 50% (Gorin et al., 1997).

Text segmentation has generally been investigated from one of two perspectives: approaches based on strong theories of discourse structure, and approaches based on lexical text cohesion (Morris and Hirst, 1991). Hearst (1997) describes a robust technique based on a vector model of lexical cohesion. Techniques based on strong discourse-models are discussed in Chapter 18 and Chapter 20.

Research on text summarization began with the work of Luhn (1958) on the automatic generation of abstracts. A collection of papers on text summarization can be found in Hovy and Radev (1998).

An important extension of the vector space model known as **Latent Semantic Indexing** (LSI) (Deerwester et al., 1990) uses the singular value decomposition method as means of *reducing the dimensionality* of vector models with the intent of discovering higher-order regularities in the original term-by-document matrix. Berry et al. (1999) present a useful review of numerical methods for dimensionality reduction in vector models. Although LSI began life as a retrieval method, it has been applied to a wide variety of applications including models of lexical acquisition (Landauer and Dumais, 1997), question answering (Jones, 1997), and most recently, essay grading (Landauer et al., 1997).

LATENT SEMANTIC INDEXING

Baeza-Yates and Ribeiro-Neto (1999) is a comprehensive text covering many of newest advances and trends in information retrieval. Frakes and Baeza-Yates (1992) is a more nuts and bolts text which includes a considerable amount of useful C code. Older classic texts include Salton and McGill (1983) and van Rijsbergen (1975). Many of the classic papers in the field can be found in Sparck Jones and Willett (1997). Current work is published in the annual proceedings of the ACM Special Interest Group on Information Retrieval (SIGIR). The periodic TREC conference proceedings contain results from standardized evaluations organized by the U.S. government. The primary journals in the field are the *Journal of the American Society of Information Sciences*, *ACM Transactions on Information Systems*, *Information Processing and Management*, and *Information Retrieval*.

EXERCISES

17.1 Collect a small corpus of example sentences of varying lengths from any newspaper or magazine. Using WordNet, or any standard dictionary, determine how many senses there are for each of the open-class words in each sentence. How many distinct combinations of senses are there for each sentence? How does this number seem to vary with sentence length?

17.2 Using WordNet, or a standard reference dictionary, tag each open-class word in your corpus with its correct tag. Was choosing the correct sense always a straightforward task. Report on any difficulties you encountered.

17.3 Using the same corpus, isolate the words taking part in all the verb-subject and verb-object relations. How often does it appear to be the case that the words taking part in these relations could be disambiguated using only information about the words in the relation?

17.4 Between the words *eat* and *find* which would you expect to be more effective in selectional restriction-based sense disambiguation? Why?

17.5 Implement and experiment with a decision-list sense disambiguation system. As a model, use the kinds of features shown in Figure 17.1. For more details on decision-list learning see Russell and Norvig (1995). To facilitate evaluation of your system, you should obtain one of the freely available sense-tagged corpora.

17.6 Using your favorite dictionary, simulate the word overlap disambiguation algorithm described on page 645 on the phrase *Time flies like an arrow*. Assume that the words are to be disambiguated one at a time, from left to right, and that the results from earlier decisions are used later in the process.

17.7 Formulate a set of detailed queries from a domain you are familiar with, and submit them to a number of popular search engines. Using a series of fixed cutoffs, assess the precision of each of these search engines.

17.8 For each of the returned documents that you judged **not relevant** in Exercise 17.7, come up with an account as to why it might have been returned.

17.9 Consider the relevant documents that were returned by some, but not all, of the search engines in Exercise 17.7. For the search engines that failed to retrieve a relevant document:

a. Determine if the search engine contains the relevant document.

b. If it does, then come up with an account for why it did not return it (or did not rank it highly).

17.10 Investigate five of the more popular search engines and determine which, if any, are employing some kind of morphological analysis.

17.11 Expand the queries used in Exercise 17.7 to include all of the morphological variants of each query word. Submit these expanded queries to your original set of search engines. Does such morphological processing seem warranted?

17.12 Using WordNet, expand your queries to include all the synonyms of all the terms in the original query. Report on the results of submitting the expanded queries to a set of search engines.

17.13 Using WordNet, expand your queries to include only those synonyms that are appropriate for each of the terms in the original query. In other words, only include synonyms for the senses of terms you intended in the original query. Submit these expanded queries to a set of search engines, and compare the results to those you achieved in the previous exercise.

17.14 Word sense disambiguation seems to have little effect on retrieval performance in settings where long queries are used. Suggest reasons for why this might be the case.

17.15 Find, or create, a collection of documents that have been separated into distinct topical categories. E-mail messages that have been manually placed into distinct folders are a good source for such a collection. Using this collection, implement and evaluate a naive Bayes approach to text classification.