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.

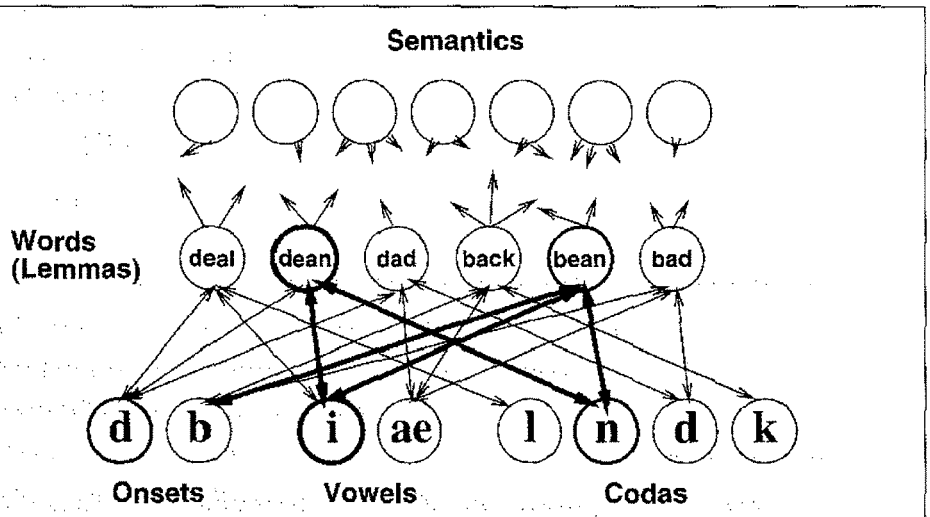

Figure 5.22 The network model of Dell (1986, 1988), showing the mechanism for lexical bias (modified from Dell (1988, p. 134)). The boldfaced nodes indicate nodes with lots of activation. The intended word *dean* has a greater chance of slipping to *bean* because of the existence of the *bean* node. 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 $c(w_{n-N+1}^{n-1}) = 0$) and our definition is complete:

$$P(w_n | w_{n-N+1}^{n-1}) = P(w_n | w_{n-N+1}^{n-N+2}) \tag{6.38}$$

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 $P(w_{\cdot\cdot})$ 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.

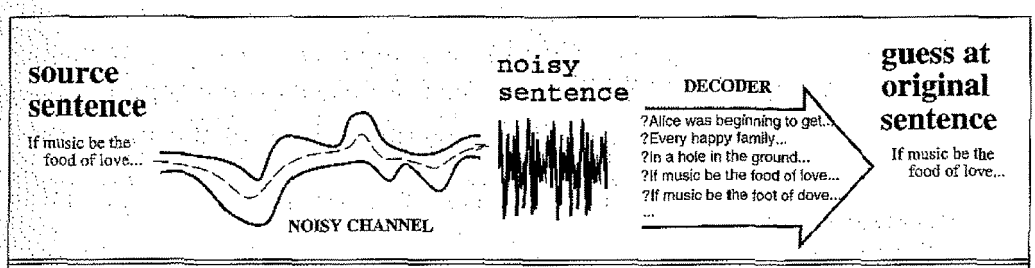

Figure 7.1 The noisy channel model applied to entire sentences (Figure 5.1 showed its application to individual words). Modern speech recognizers work by searching through a huge space of potential "source" sentences and choosing the one which has the highest probability of generating the "noisy" sentence. 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

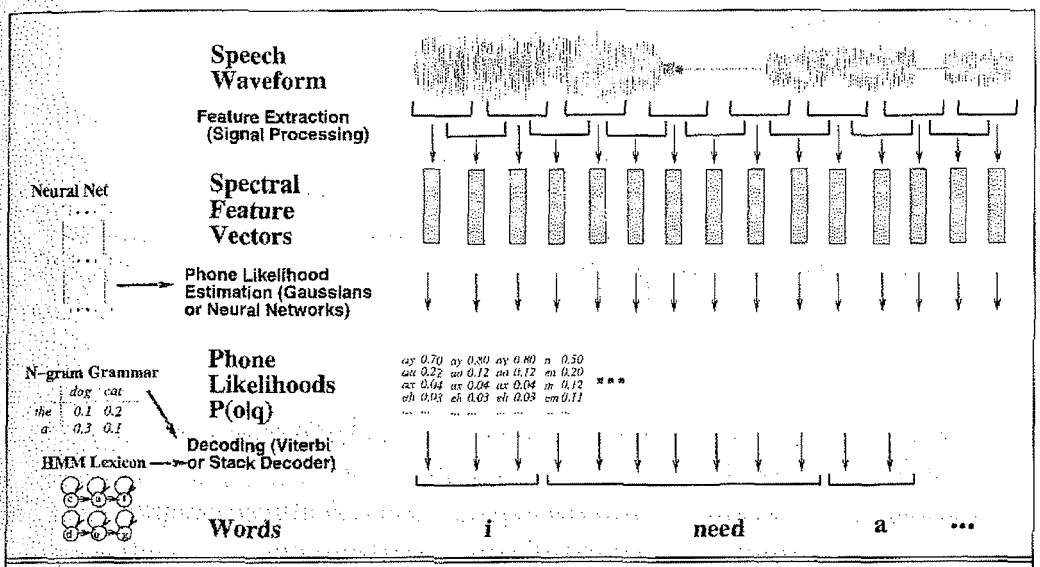

Figure 7.2 Schematic architecture for a (simplified) speech recognizer.

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

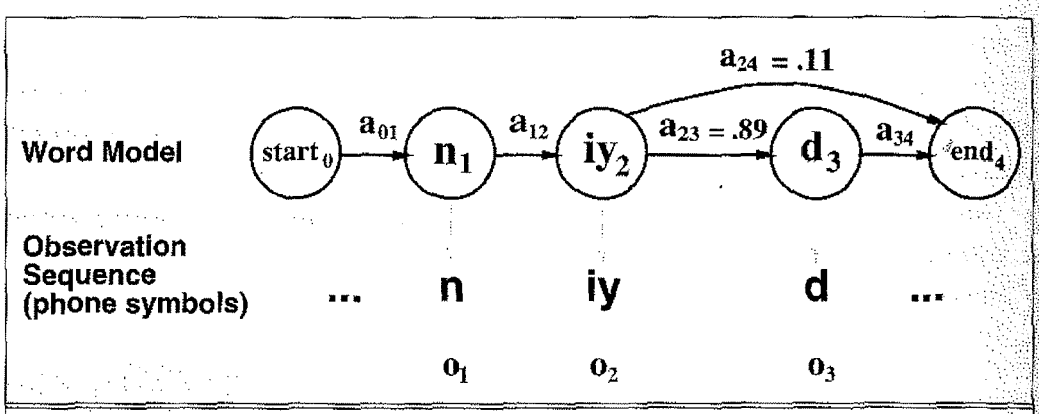

Figure 7.3 A simple weighted automaton or Markov chain pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. The transition probabilities a_{xy} between two states x and y are 1.0 unless otherwise specified.

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.

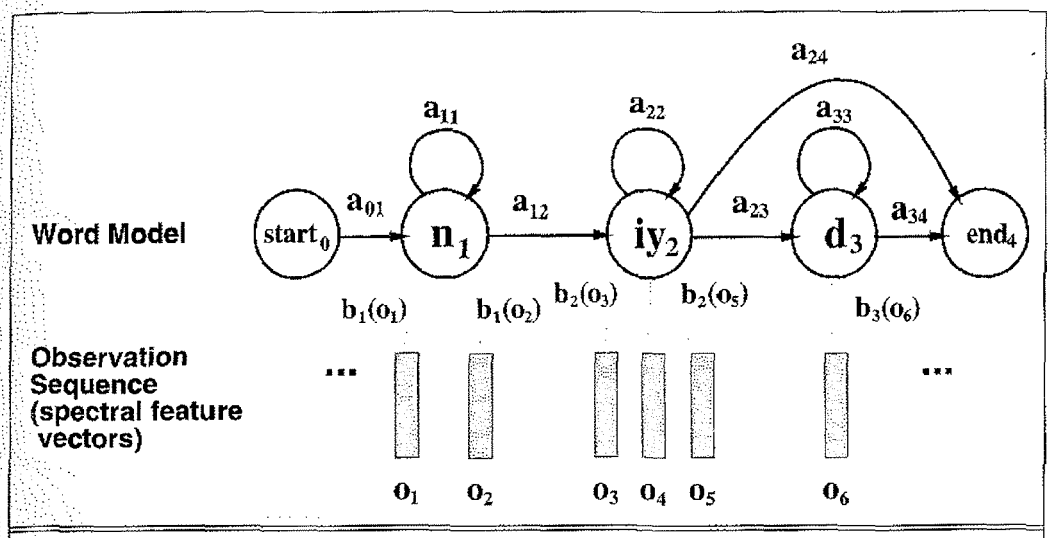

Figure 7.4 An HMM pronunciation network for the word *need*, showing the transition probabilities, and a sample observation sequence. Note the addition of the output probabilities B. HMMs used in speech recognition usually use self-loops on the states to model variable phone durations.

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*.

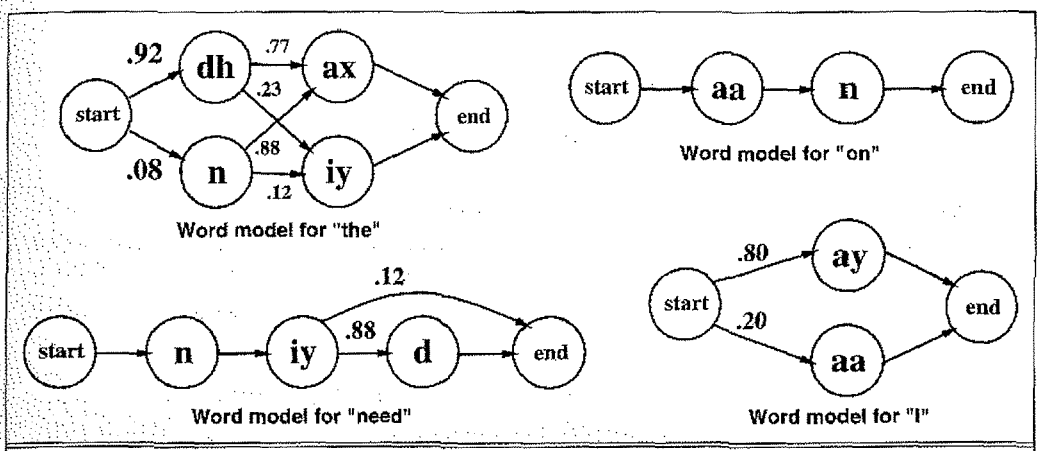

Figure 7.5 Pronunciation networks for the words *I*, *on*, *need*, and *the*. All networks (especially *the*) are significantly simplified.

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.

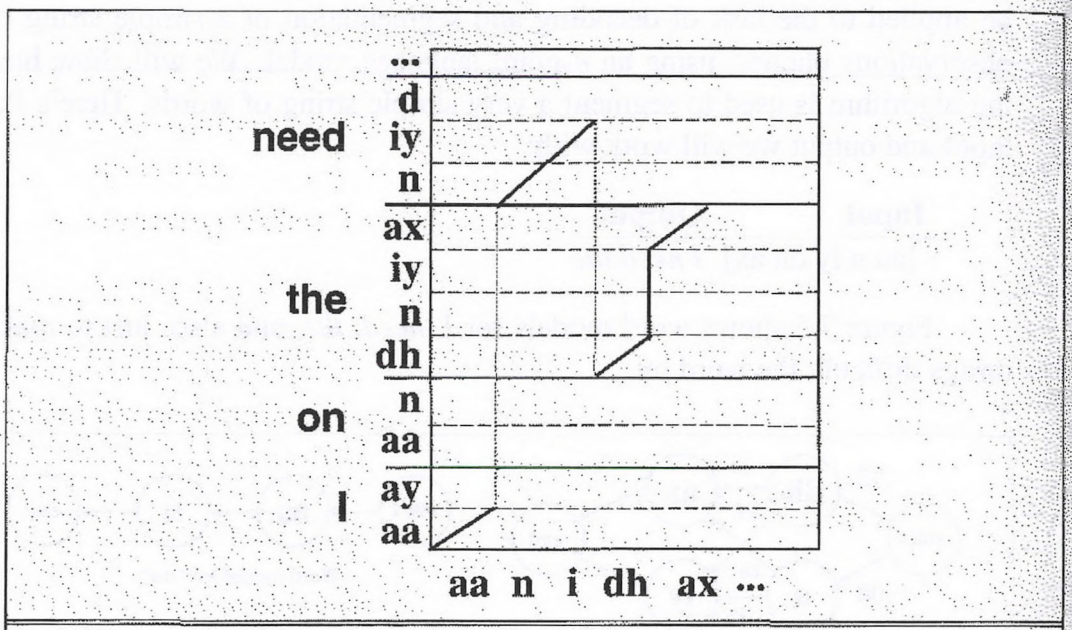

Figure 7.6 An illustration of the results of the Viterbi algorithm used to find the most-likely phone sequence (and hence estimate the most-likely word sequence).

More formally, we are searching for the best state sequence $q^* = (q_1 q_2 \ldots q_T)$, given an observation sequence $o = (o_1 o_2 \ldots o_T)$ 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 $t-1$:

$$viterbi[t,i] = \max_{q_1,q_2,\cdots,q_{t-1}} P(q_1 q_2 \ldots q_{t-1}, q_t = i, o_1, o_2 \ldots o_t | \lambda) \qquad (7.8)$$

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

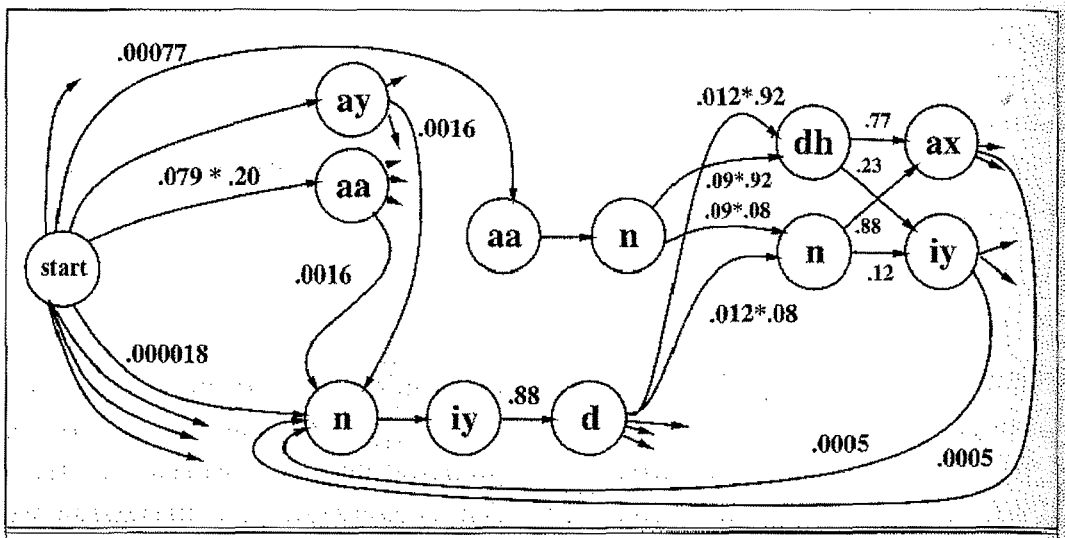

Figure 7.8 Single automaton made from the words *I*, *need*, *on*, and *the*. The arcs between words have probabilities computed from Figure 7.7. For lack of space the figure only shows a few of the between-word arcs.

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.

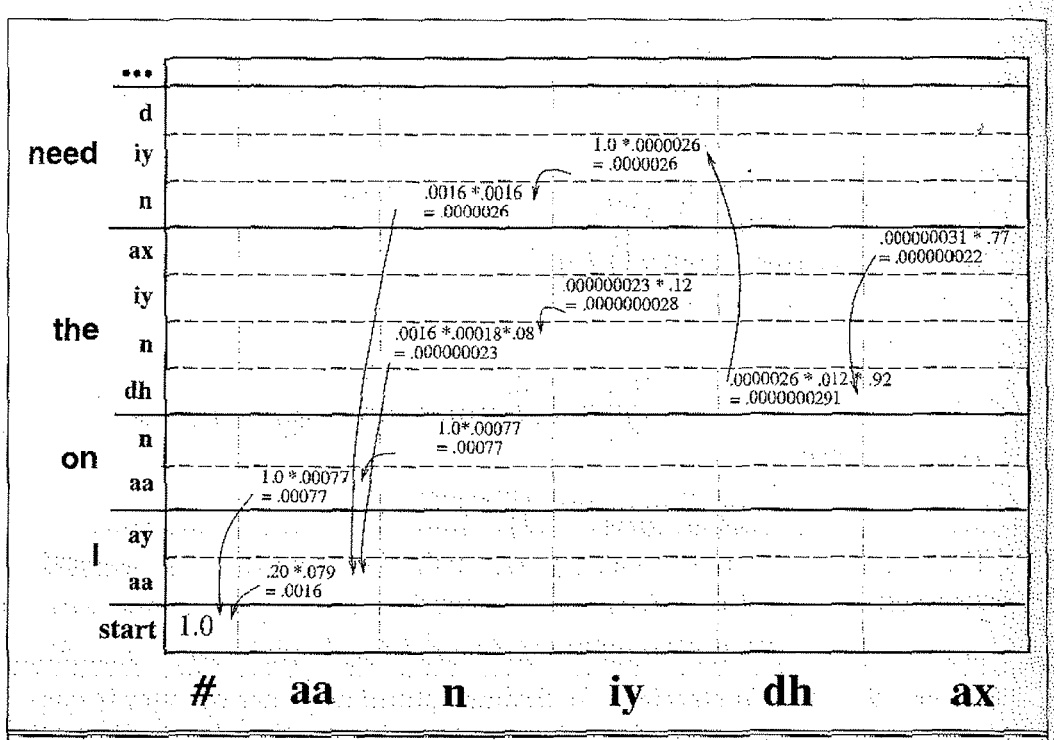

Figure 7.10 The entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. Backtracing from the successful last word (*the*), we can reconstruct the word sequence *I need the*.

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).

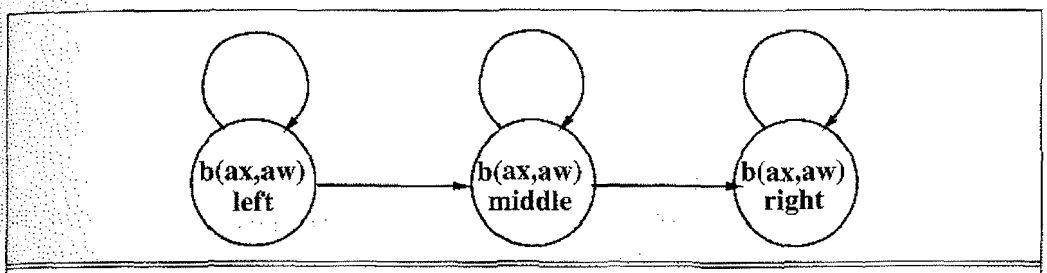

Figure 7.11 An example of the context-dependent triphone b(ax,aw) (the phone [b] preceded by a [ax] and followed by a [aw], as in the beginning of *about*, showing its left, middle, and right subphones.

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

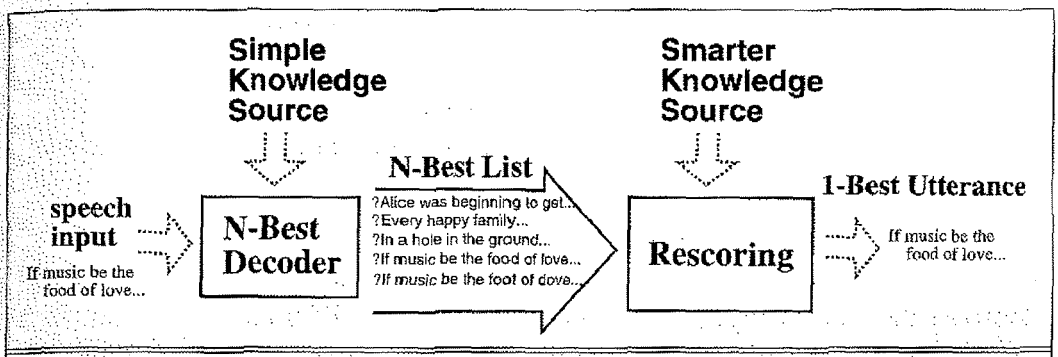

Figure 7.12 The use of N-best decoding as part of a two-stage decoding model. Efficient but unsophisticated knowledge sources are used to return the N-best utterances. This significantly reduces the search space for the second pass models, which are thus free to be very sophisticated but slow.

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

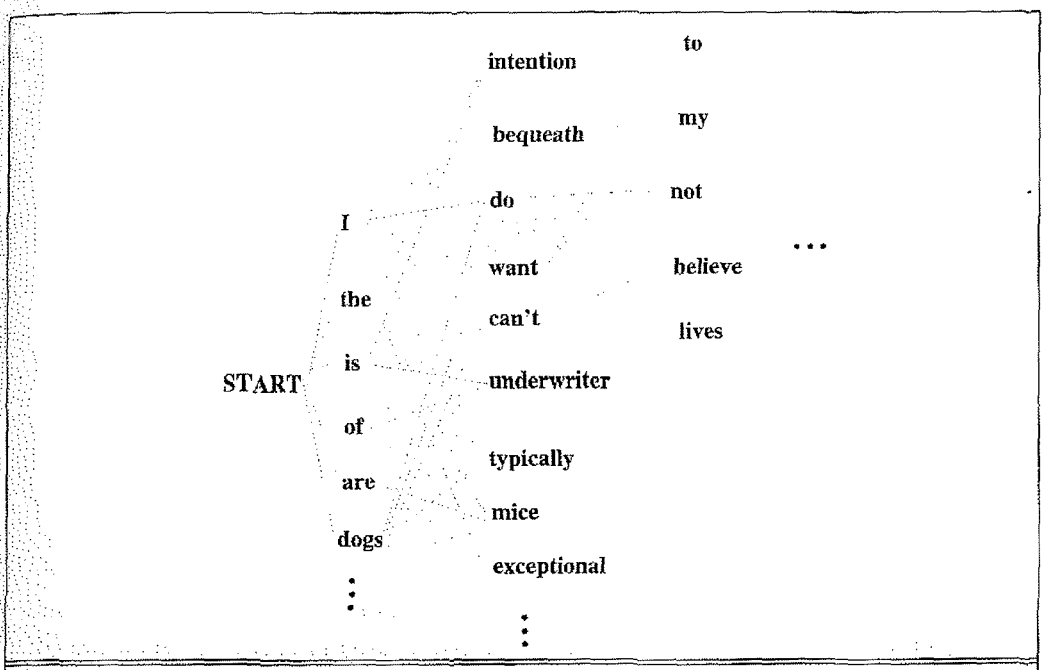

Figure 7.13 A visual representation of the implicit lattice of allowable word sequences that defines a language. The set of sentences of a language is far too large to represent explicitly, but the lattice gives a metaphor for exploring substrings of these sentences.

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.

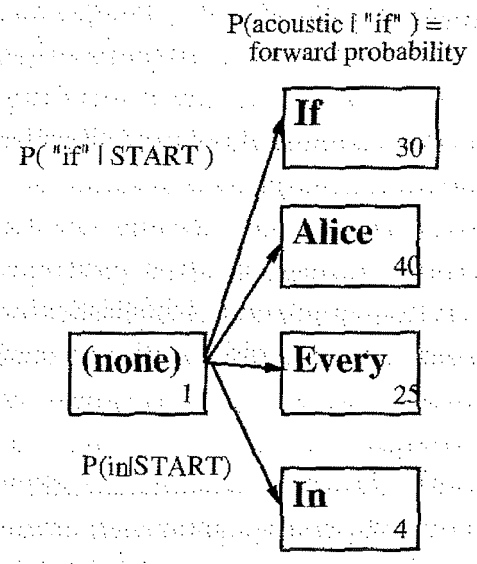

Figure 7.15 The beginning of the search for the sentence *If music be the food of love*. At this early stage *Alice* is the most likely hypothesis. (It has a higher score than the other hypotheses.)

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.

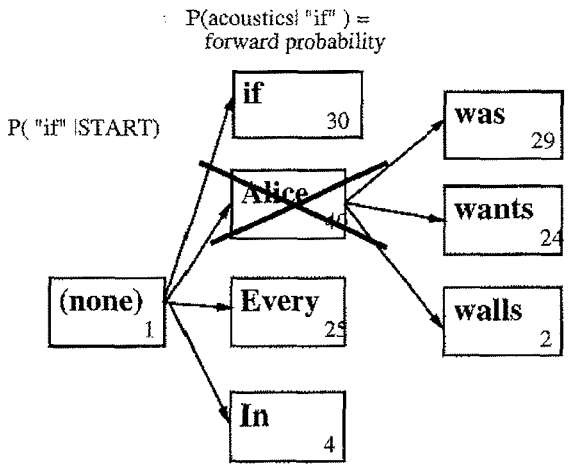

Figure 7.16 The next step of the search for the sentence *If music be the food of love*. We've now expanded the *Alice* node and added three extensions which have a relatively high score (*was*, *wants*, and *walls*). Note that now the node with the highest score is *START if*, which is not along the *START Alice* path at all!

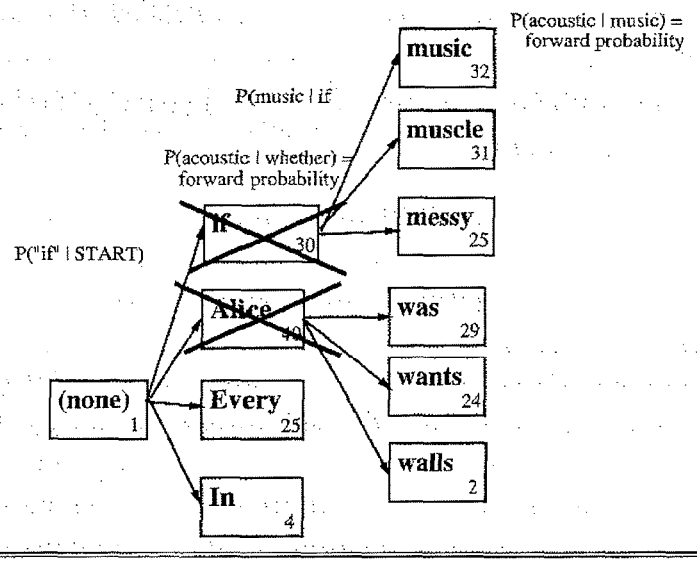

Figure 7.17 We've now expanded the *if* node. The hypothesis *START if music* currently has the highest score.

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.

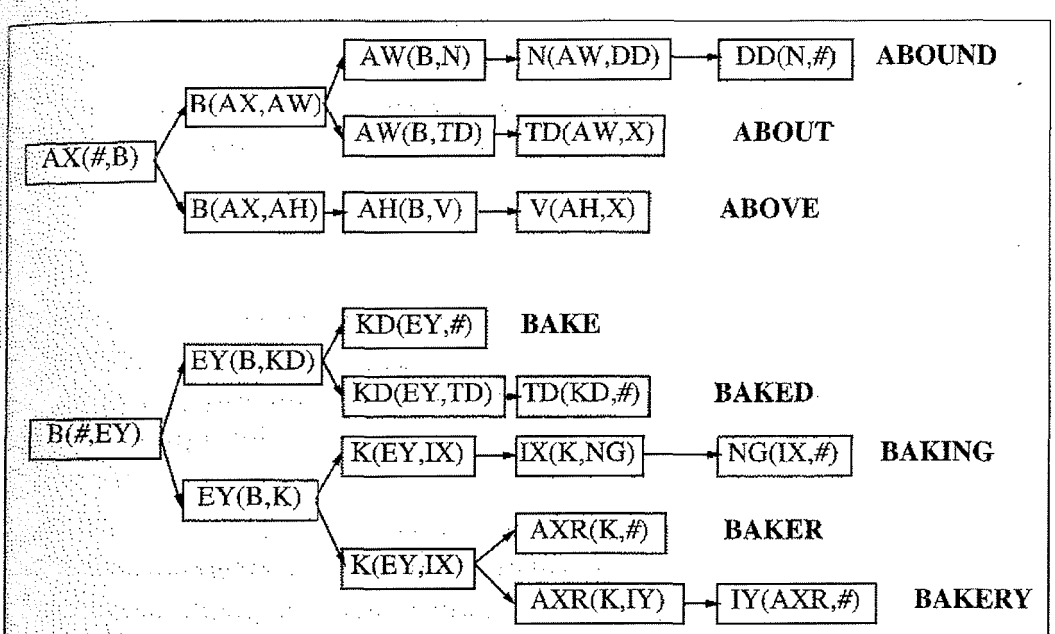

Figure 7.18 A tree-structured lexicon from the Sphinx-II recognizer (after Ravishankar (1996)). Each node corresponds to a particular triphone in a slightly modified version of the ARPAbet; thus EY(B,KD) means the phone EY preceded by a B and followed by the closure of a K.

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".

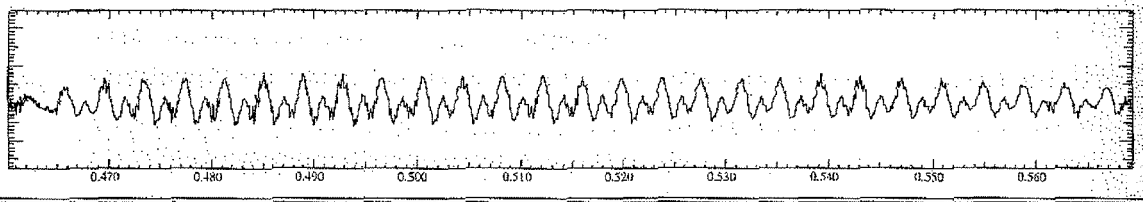

Figure 7.19 A waveform of the vowel [iy] from the utterance shown in Figure 7.20. The *y*-axis shows the changes in air pressure above and below normal atmospheric pressure. The *x*-axis shows time. Notice that the wave repeats regularly.

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.

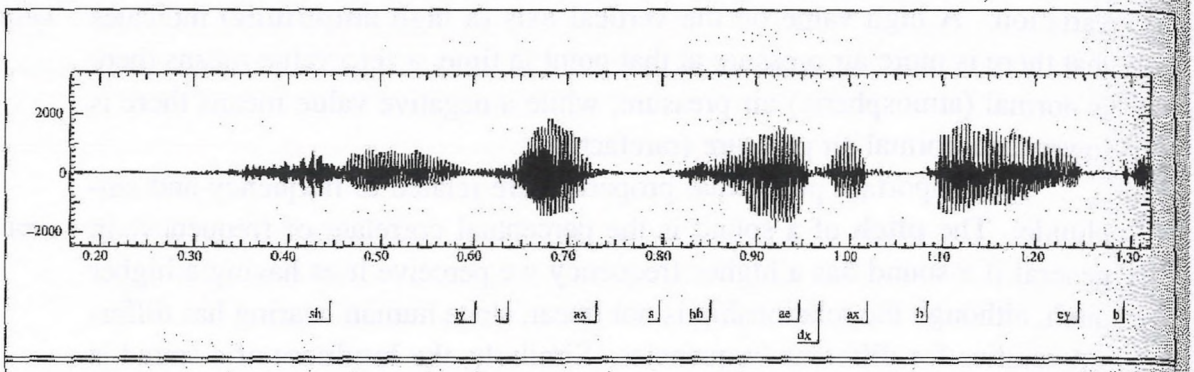

Figure 7.20 A waveform of the sentence "She just had a baby" from the Switchboard corpus (conversation 4325). The speaker is female, was 20 years old in 1991, which is approximately when the recording was made, and speaks the South Midlands dialect of American English. The phone labels show where each 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

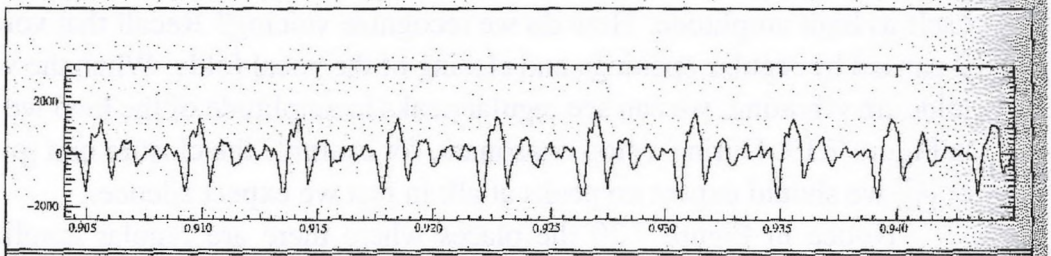

Figure 7.21 The waveform of part of the vowel [æ] from the word *had* cut out from the waveform shown in Figure 7.20.

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

LPC

SPECTRAL PEAKS

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).

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..

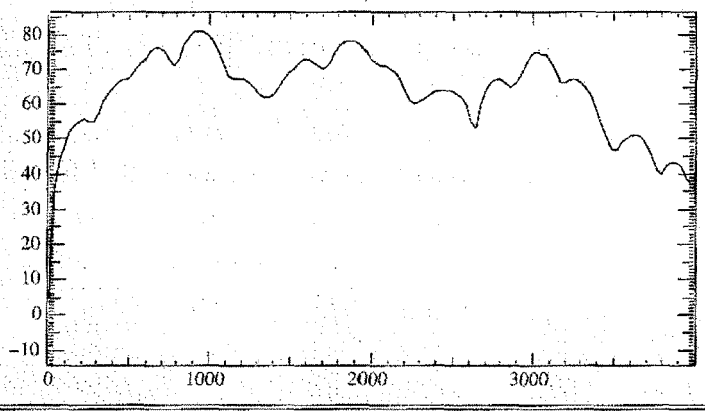

Figure 7.22 An LPC spectrum for the vowel [æ] waveform of *She just had a baby* at the point in time shown in Figure 7.21. LPC makes it easy to see **formants**.

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

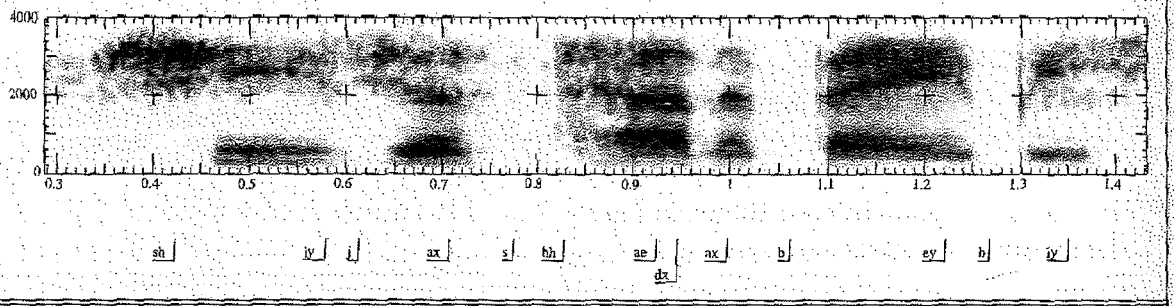

Figure 7.23 A spectrogram of the sentence "She just had a baby" whose waveform was shown in Figure 7.20. One way to think of a spectrogram is as a collection of spectra (time-slices) like Figure 7.22 placed end to end.

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.

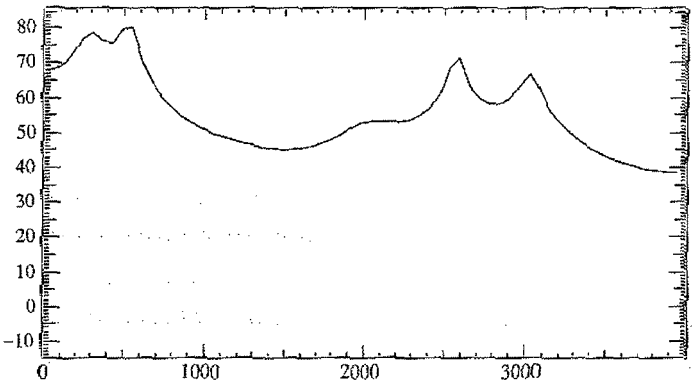

Figure 7.24 A smoothed (LPC) spectrum for the vowel [iy] at the start of *She just had a baby*. Note that the first formant (540 Hz) is much lower than the first formant for [æ] shown in Figure 7.22, while the second formant (2581 Hz) is much higher than the second formant for [æ].

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.

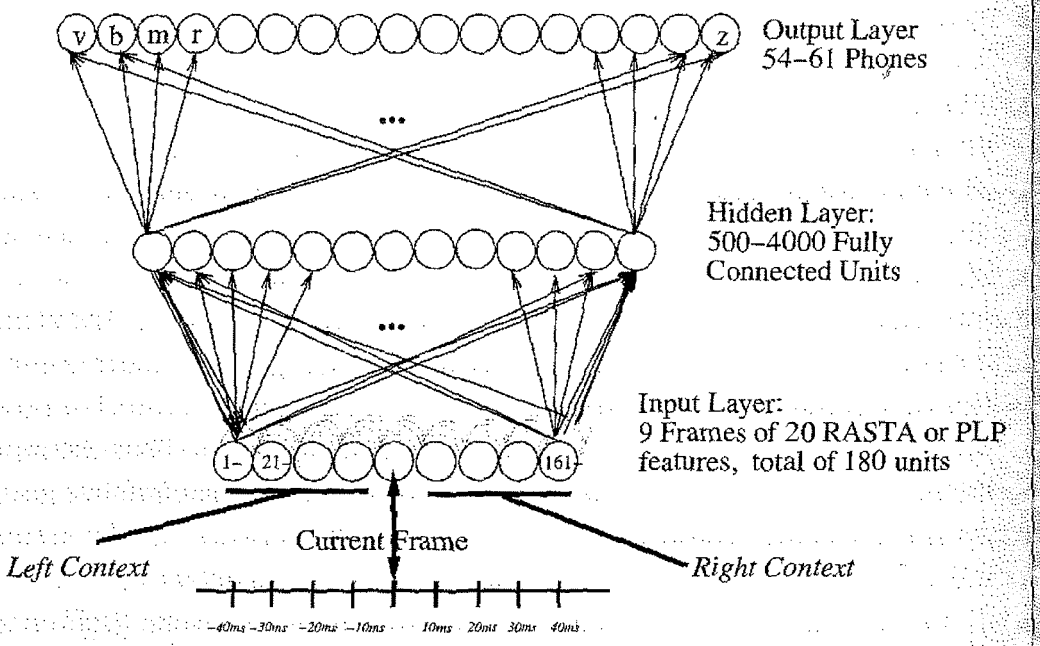

Figure 7.25 A neural net used to estimate phone state probabilities. Such a net can be used in an HMM model as an alternative to the Gaussian models. This particular net is from the MLP systems described in Bourlard and Morgan (1994); it is given a vector of features for a frame and for the four frames 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.

TARGET

This specification is often called the **target**, as it is this that we want the synthesizer to produce.

WAVEFORM CONCATENATION

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.

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 pro-

DIPHONES

hibitive 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

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 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.