the relative likelihoods for the frequently occurring values. This estimation technique is called the **expected likelihood estimator (ELE)**. To see the difference between this and the MLE, consider a word w that happens not to occur in the corpus, and consider estimating the probability that w occurs in one of 40 word classes L_1, ..., L_{40}. Thus we have a random variable X, where $X = x_i$ only if w appears in word category L_i. The MLE for $PROB(X = x_i)$ will not be defined because the formula has a zero denominator. The ELE, however, gives an equally likely probability to each possible word class. With 40 word classes, for instance, each V_i will be .5, and thus $PROB(L_i \mid w) \cong .5/20 = .025$. This estimate better reflects the fact that we have no information about the word. On the other hand, the ELE is very conservative. If w appears in the corpus five times, once as a verb and four times as a noun, then the MLE estimate of $PROB(N \mid w)$ would be .8, while the ELE estimate would be 4.5/25 = .18, a very small value compared to intuition.

Evaluation

Once you have a set of estimated probabilities and an algorithm for some particular application, you would like to be able to tell how well your new technique performs compared with other algorithms or variants of your algorithm. The general method for doing this is to divide the corpus into two parts: the **training set** and the **test set**. Typically, the test set consists of 10–20 percent of the total data. The training set is then used to estimate the probabilities, and the algorithm is run on the test set to see how well it does on new data. Running the algorithm on the training set is not considered a reliable method of evaluation because it does not measure the generality of your technique. For instance, you could do well on the training set simply by remembering all the answers and repeating them back in the test! A more thorough method of testing is called **cross-validation**, which involves repeatedly removing different parts of the corpus as the test set, training on the remainder of the corpus, and then evaluating on the new test set. This technique reduces the chance that the test set selected was somehow easier than you might expect.

7.3 Part-of-Speech Tagging

Part-of-speech tagging involves selecting the most likely sequence of syntactic categories for the words in a sentence. A typical set of tags, used in the Penn Treebank project, is shown in Figure 7.3. In Section 7.1 you saw the simplest algorithm for this task: Always choose the interpretation that occurs most frequently in the training set. Surprisingly, this technique often obtains about a 90 percent success rate, primarily because over half the words appearing in most corpora are not ambiguous. So this measure is the starting point from which to evaluate algorithms that use more sophisticated techniques. Unless a method does significantly better than 90 percent, it is not working very well.

1. CC	Coordinating conjunction	19. PP$	Possessive pronoun
2. CD	Cardinal number	20. RB	Adverb
3. DT	Determiner	21. RBR	Comparative adverb
4. EX	Existential *there*	22. RBS	Superlative Adverb
5. FW	Foreign word	23. RP	Particle
6. IN	Preposition / subord. conj	24. SYM	Symbol (math or scientific)
7. JJ	Adjective	25. TO	to
8. JJR	Comparative adjective	26. UH	Interjection
9. JJS	Superlative adjective	27. VB	Verb, base form
10. LS	List item marker	28. VBD	Verb, past tense
11. MD	Modal	29. VBG	Verb, gerund/pres. participle
12. NN	Noun, singular or mass	30. VBN	Verb, past participle
13. NNS	Noun, plural	31. VBP	Verb, non-3s, present
14. NNP	Proper noun, singular	32. VBZ	Verb, 3s, present
15. NNPS	Proper noun, plural	33. WDT	Wh-determiner
16. PDT	Predeterminer	34. WP	Wh-pronoun
17. POS	Possessive ending	35. WPZ	Possessive wh-pronoun
18. PRP	Personal pronoun	36. WRB	Wh-adverb

Figure 7.3 The Penn Treebank tagset

The general method to improve reliability is to use some of the local context of the sentence in which the word appears. For example, in Section 7.1 you saw that choosing the verb sense of *flies* in the sample corpus was the best choice and would be right about 60 percent of the time. If the word is preceded by the word *the*, on the other hand, it is much more likely to be a noun. The technique developed in this section is able to exploit such information.

Consider the problem in its full generality. Let $w_1, ..., w_T$ be a sequence of words. We want to find the sequence of lexical categories $C_1, ..., C_T$ that maximizes

1. $PROB(C_1, ..., C_T \mid w_1, ..., w_T)$

Unfortunately, it would take far too much data to generate reasonable estimates for such sequences, so direct methods cannot be applied. There are, however, reasonable approximation techniques that produce good results. To develop them, you must restate the problem using Bayes' rule, which says that this conditional probability equals

2. $(PROB(C_1, ..., C_T) * PROB(w_1, ..., w_T \mid C_1, ..., C_T)) / PROB(w_1, ..., w_T)$

As before, since we are interested in finding the $C_1, ..., C_n$ that gives the maximum value, the common denominator in all these cases will not affect the answer. Thus the problem reduces to finding the sequence $C_1, ..., C_n$ that maximizes the formula

3. $PROB(C_1, ..., C_T) * PROB(w_1, ..., w_T \mid C_1, ..., C_T)$

There are still no effective methods for calculating the probability of these long sequences accurately, as it would require far too much data. But the probabilities can be approximated by probabilities that are simpler to collect by making some independence assumptions. While these independence assumptions are not really valid, the estimates appear to work reasonably well in practice. Each of the two expressions in formula 3 will be approximated. The first expression, the probability of the sequence of categories, can be approximated by a series of probabilities based on a limited number of previous categories. The most common assumptions use either one or two previous categories. The **bigram** model looks at pairs of categories (or words) and uses the conditional probability that a category C_i will follow a category C_{i-1}, written as $PROB(C_i \mid C_{i-1})$. The **trigram** model uses the conditional probability of one category (or word) given the two preceding categories (or words), that is, $PROB(C_i \mid C_{i-2}\ C_{i-1})$. These models are called **n-gram** models, in which n represents the number of words used in the pattern. While the trigram model will produce better results in practice, we will consider the bigram model here for simplicity. Using bigrams, the following approximation can be used:

$$PROB(C_1, ..., C_T) \cong \prod_{i=1,T} PROB(C_i \mid C_{i-1})$$

To account for the beginning of a sentence, we posit a pseudocategory ø at position 0 as the value of C_0. Thus the first bigram for a sentence beginning with an ART would be $PROB(\text{ART} \mid ø)$. Given this, the approximation of the probability of the sequence ART N V N using bigrams would be

$$PROB(\text{ART N V N}) \cong PROB(\text{ART} \mid ø) * PROB(\text{N} \mid \text{ART}) * PROB(\text{V} \mid \text{N}) * PROB(\text{N} \mid \text{V})$$

The second probability in formula 3,

$$PROB(w_1, ..., w_T \mid C_1, ..., C_T)$$

can be approximated by assuming that a word appears in a category independent of the words in the preceding or succeeding categories. It is approximated by the product of the probability that each word occurs in the indicated part of speech, that is, by

$$PROB(w_1, ..., w_T \mid C_1, ..., C_T) \cong \prod_{i=1,T} PROB(w_i \mid C_i)$$

With these two approximations, the problem has changed into finding the sequence $C_1, ..., C_T$ that maximizes the value of

$$\prod_{i=1,T} PROB(C_i \mid C_{i-1}) * PROB(w_i \mid C_i)$$

The advantage of this new formula is that the probabilities involved can be readily estimated from a corpus of text labeled with parts of speech. In particular, given a database of text, the bigram probabilities can be estimated simply by counting the number of times each pair of categories occurs compared to the

Category	Count at i	Pair	Count at i,i+1	Bigram	Estimate
ø	300	ø, ART	213	*PROB*(ART | ø)	.71
ø	300	ø, N	87	*PROB*(N | ø)	.29
ART	558	ART, N	558	*PROB*(N | ART)	1
N	833	N, V	358	*PROB*(V | N)	.43
N	833	N, N	108	*PROB*(N | N)	.13
N	833	N, P	366	*PROB*(P | N)	.44
V	300	V, N	75	*PROB*(N | V)	.35
V	300	V, ART	194	*PROB*(ART | V)	.65
P	307	P, ART	226	*PROB*(ART | P)	.74
P	307	P, N	81	*PROB*(N | P)	.26

Figure 7.4 Bigram probabilities from the generated corpus

individual category counts. The probability that a V follows an N would be estimated as follows:

$$PROB(C_i{=}V \mid C_{i-1}{=}N) \cong \frac{\text{Count(N at position i–1 and V at i)}}{\text{Count(N at position i–1)}}$$

Figure 7.4 gives some bigram frequencies computed from an artificially generated corpus of simple sentences. The corpus consists of 300 sentences but has words in only four categories: N, V, ART, and P. In contrast, a typical real tagset used in the Penn Treebank, shown in Figure 7.3, contains about 40 tags. The artificial corpus contains 1998 words: 833 nouns, 300 verbs, 558 articles, and 307 prepositions. Each bigram is estimated using the previous formula. To deal with the problem of sparse data, any bigram that is not listed here will be assumed to have a token probability of .0001.

The lexical-generation probabilities, $PROB(w_i \mid C_i)$, can be estimated simply by counting the number of occurrences of each word by category. Figure 7.5 gives some counts for individual words from which the lexical-generation probability estimates in Figure 7.6 are computed. Note that the lexical-generation probability is the probability that a given category is realized by a specific word, not the probability that a given word falls in a specific category. For instance, *PROB*(*the* | ART) is estimated by Count (# times *the* is an ART) / Count (# times an ART occurs). The other probability, *PROB*(ART | *the*), would give a very different value.

Given all these probability estimates, how might you find the sequence of categories that has the highest probability of generating a specific sentence? The brute force method would be to generate all possible sequences that could generate the sentence and then estimate the probability of each and pick the best one. The problem with this is that there are an exponential number of sequences—given N categories and T words, there are N^T possible sequences.

	N	V	ART	P	TOTAL
flies	21	23	0	0	44
fruit	49	5	1	0	55
like	10	30	0	21	61
a	1	0	201	0	202
the	1	0	300	2	303
flower	53	15	0	0	68
flowers	42	16	0	0	58
birds	64	1	0	0	65
others	592	210	56	284	1142
TOTAL	833	300	558	307	1998

Figure 7.5 A summary of some of the word counts in the corpus

PROB(*the* | ART)	.54	*PROB*(*a* | ART)	.360
PROB(*flies* | N)	.025	*PROB*(*a* | N)	.001
PROB(*flies* | V)	.076	*PROB*(*flower* | N)	.063
PROB(*like* | V)	.1	*PROB*(*flower* | V)	.05
PROB(*like* | P)	.068	*PROB*(*birds* | N)	.076
PROB(*like* | N)	.012		

Figure 7.6 The lexical-generation probabilities

Luckily, you can do much better than this because of the independence assumptions that were made about the data.

Since we are only dealing with bigram probabilities, the probability that the i'th word is in a category C_i depends only on the category of the (i–1)th word, C_{i-1}. Thus the process can be modeled by a special form of probabilistic finite state machine, as shown in Figure 7.7. Each node represents a possible lexical category and the transition probabilities (the bigram probabilities in Figure 7.4) indicate the probability of one category following another.

With such a network you can compute the probability of any sequence of categories simply by finding the path through the network indicated by the sequence and multiplying the transition probabilities together. For instance, the sequence ART N V N would have the probability .71 * 1 * .43 * .35 = .107. This representation, of course, is only accurate if the probability of a category occurring depends only on the one category before it. In probability theory this is often called the **Markov assumption**, and networks like that in Figure 7.7 are called **Markov chains**.

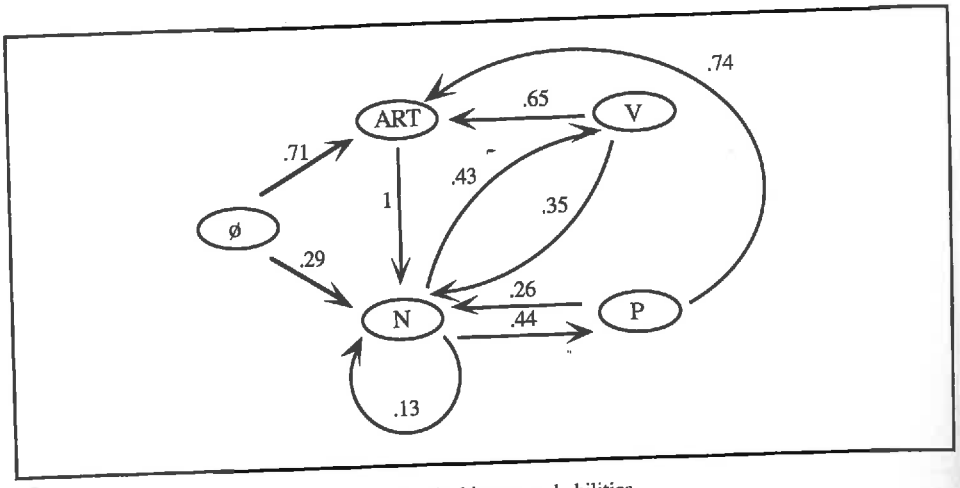

Figure 7.7 A Markov chain capturing the bigram probabilities

The network representation can now be extended to include the lexical-generation probabilities as well. In particular, we allow each node to have an **output probability**, which gives a probability to each possible output that could correspond to the node. For instance, node N in Figure 7.7 would be associated with a probability table that indicates, for each word, how likely that word is to be selected if we randomly select a noun. The output probabilities are exactly the lexical-generation probabilities shown in Figure 7.6. A network like that in Figure 7.7 with output probabilities associated with each node is called a **Hidden Markov Model** (HMM). The word *hidden* in the name indicates that for a specific sequence of words, it is not clear what state the Markov model is in. For instance, the word *flies* could be generated from state N with a probability of .025 (given the values in Figure 7.6), or it could be generated from state V with a probability .076. Because of this ambiguity, it is no longer trivial to compute the probability of a sequence of words from the network. If you are given a particular sequence, however, the probability that it generates a particular output is easily computed by multiplying the probabilities on the path times the probabilities for each output. For instance, the probability that the sequence N V ART N generates the output *Flies like a flower* is computed as follows. The probability of the path N V ART N, given the Markov model in Figure 7.7, is .29 * .43 * .65 * 1 = .081. The probability of the output being *Flies like a flower* for this sequence is computed from the output probabilities given in Figure 7.6:

$$\begin{aligned} &PROB(flies \mid \text{N}) * PROB(like \mid \text{V}) * PROB(a \mid \text{ART}) * PROB(flower \mid \text{N}) \\ &\quad = .025 * .1 * .36 * .063 \\ &\quad = 5.4 * 10^{-5} \end{aligned}$$

Multiplying these together gives us the likelihood that the HMM would generate the sentence, $4.37 * 10^{-6}$. More generally, the formula for computing the probability of a sentence $w_1, ..., w_T$ given a sequence $C_1, ..., C_T$ is

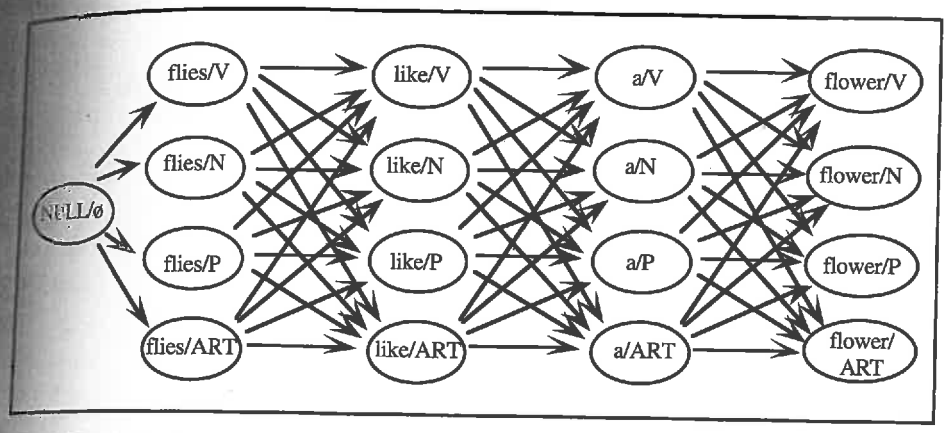

Figure 7.8 Encoding the 256 possible sequences exploiting the Markov assumption

$$\Pi_{i=1,T} PROB(C_i \mid C_{i-1}) * PROB(w_i \mid C_i)$$

Now we can resume the discussion of how to find the most likely sequence of categories for a sequence of words. The key insight is that because of the Markov assumption, you do not have to enumerate all the possible sequences. In fact, sequences that end in the same category can be collapsed together since the next category only depends on the one previous category in the sequence. So if you just keep track of the most likely sequence found so far for each possible ending category, you can ignore all the other less likely sequences. For example, consider the problem of finding the most likely categories for the sentence *Flies like a flower,* with the lexical-generation probabilities and bigram probabilities discussed so far. Given that there are four possible categories, there are $4^4 = 256$ different sequences of length four. The brute force algorithm would have to generate all 256 sequences and compare their probabilities in order to find this one. Exploiting the Markov assumption, however, this set of sequences can be collapsed into a representation that considers only the four possibilities for each word. This representation, shown as a transition diagram in Figure 7.8, represents all 256 sequences. To find the most likely sequence, you sweep forward through the words one at a time finding the most likely sequence for each ending category. In other words, you first find the four best sequences for the two words *flies like*: the best ending with *like* as a V, the best as an N, the best as a P, and the best as an ART. You then use this information to find the four best sequences for the three words *flies like a,* each one ending in a different category. This process is repeated until all the words are accounted for. This algorithm is usually called the **Viterbi** algorithm. For a problem involving T words and N lexical categories, it is guaranteed to find the most likely sequence using $k*T*N^2$ steps, for some constant k, significantly better than the N^T steps required by the brute force search! The rest of this section develops the Viterbi algorithm in detail.

Given word sequence $w_1, ..., w_T$, lexical categories $L_1, ..., L_N$, lexical probabilities $PROB(w_t \mid L_i)$, and bigram probabilities $PROB(L_i \mid L_j)$, find the most likely sequence of lexical categories $C_1, ..., C_T$ for the word sequence.

Initialization Step

For i = 1 to N do
    SEQSCORE(i, 1) = $PROB(w_1 \mid L_i)$ * $PROB(L_i \mid \emptyset)$
    BACKPTR(i, 1) = 0

Iteration Step

For t = 2 to T
    For i = 1 to N
        SEQSCORE(i, t) = $MAX_{j=1,N}$(SEQSCORE(j, t–1) * $PROB(L_i \mid L_j)$) * $PROB(w_t \mid L_i)$
        BACKPTR(i, t) = index of j that gave the max above

Sequence Identification Step

C(T) = i that maximizes SEQSCORE(i, T)
For i = T–1 to 1 do
    C(i) = BACKPTR(C(i+1), i+1)

Figure 7.9 The Viterbi algorithm

The Viterbi Algorithm

We will track the probability of the best sequence leading to each possible category at each position using an N×T array, where N is the number of lexical categories ($L_1, ..., L_N$) and T is the number of words in the sentence ($w_1, ..., w_T$). This array, SEQSCORE(n, t), records the probability for the best sequence up to position t that ends with a word in category L_n. To record the actual best sequence for each category at each position, it suffices to record only the one preceding category for each category and position. Another N×T array, BACKPTR, will indicate for each category in each position what the preceding category is in the best sequence at position t–1. The algorithm, shown in Figure 7.9, operates by computing the values for these two arrays.

Let's assume you have analyzed a corpus and obtained the bigram and lexical-generation probabilities in Figures 7.4 and 7.6, and assume that any bigram probability not in Figure 7.4 has a value of .0001. Using these probabilities, the algorithm running on the sentence *Flies like a flower* will operate as follows.

The first row is set in the initialization phase using the formula

SEQSCORE(i, 1) = $PROB(Flies \mid L_i)$ * $PROB(L_i \mid \emptyset)$

where L_i ranges over V, N, ART, and P. Because of the lexical-generation probabilities in Figure 7.6, only the entries for a noun and a verb are greater than zero.

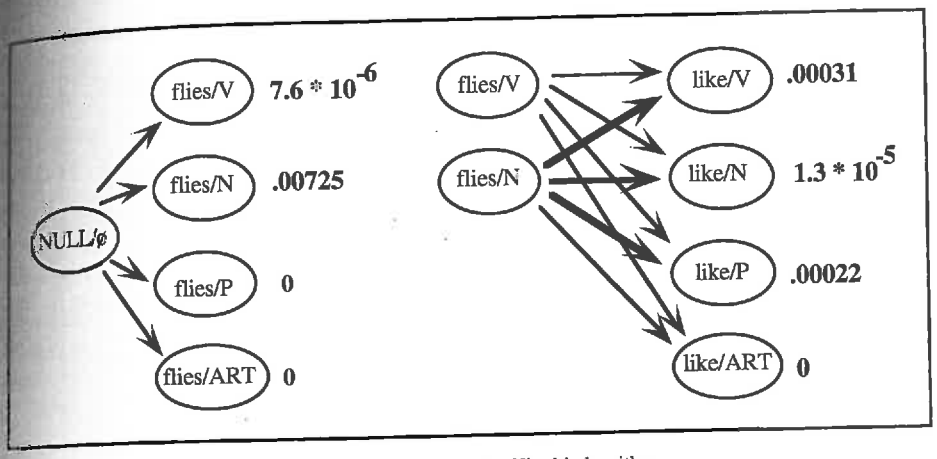

Figure 7.10 The results of the first two steps of the Viterbi algorithm

Thus the most likely sequence of one category ending in a V (= L_1) to generate *flies* has a score of $7.6 * 10^{-6}$, whereas the most likely one ending in an N (= L_2) has a score of .00725.

The result of the first step of the algorithm is shown as the left-hand side network in Figure 7.10. The probability of *flies* in each category has been computed. The second phase of the algorithm extends the sequences one word at a time, keeping track of the best sequence found so far to each category. For instance, the probability of the state like/V is computed as follows:

$$\begin{aligned} PROB(\text{like/V}) &= \text{MAX}(PROB(\text{flies/N}) * PROB(\text{V} \mid \text{N}), \\ &\qquad PROB(\text{flies/V}) * PROB(\text{V} \mid \text{V})) * \\ &\qquad PROB(\text{like/V}) \\ &= \text{MAX}(.00725 * .43, 7.6 * 10^{-6} * .0001) * .1 \\ &= 3.12 * 10^{-4} \end{aligned}$$

The difference in this value from that shown in Figure 7.10 is simply a result of the fact that the calculation here used truncated approximate values for the probabilities. In other words, the most likely sequence of length two generating *Flies like* and ending in a V has a score of $3.1 * 10^{-4}$ (and is the sequence N V), the most likely one ending in a P has a score of $2.2 * 10^{-5}$ (and is the sequence N P), and the most likely one ending in an N has a score of $1.3 * 10^{-5}$ (and is the sequence N N). The heavier arrows indicate the best sequence leading up to each node. The computation continues in the same manner until each word has been processed. Figure 7.11 shows the result after the next iteration, and Figure 7.12 shows the final result. The highest probability sequence ends in state flower/N. It is simple to trace back from this category (using BACKPTR(1, 4) and so on) to find the full sequence N V ART N, agreeing with our intuitions.

Algorithms like this can perform effectively if the probability estimates are computed from a large corpus of data that is of the same style as the input to be classified. Researchers consistently report labeling with 95 percent or better

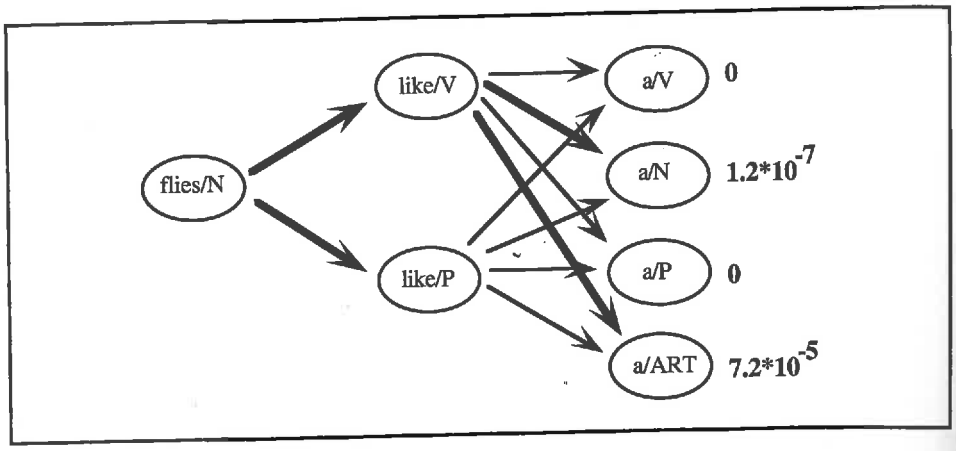

Figure 7.11 The result after the second iteration

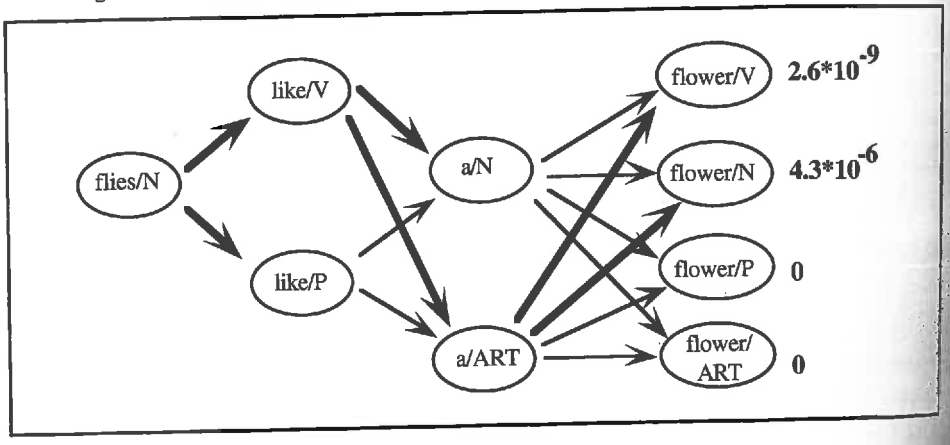

Figure 7.12 The result after the third iteration

accuracy using trigram models. Remember, however, that the naive algorithm picks the most likely category about 90 percent of the time. Still, the error rate is cut in half by introducing these techniques.

7.4 Obtaining Lexical Probabilities

Corpus-based methods suggest some new ways to control parsers. If we had some large corpora of parsed sentences available, we could use statistical methods to identify the common structures in English and favor these in the parsing algorithm. This might allow us to choose the most likely interpretation when a sentence is ambiguous, and might lead to considerably more efficient parsers that are nearly deterministic. Such corpora of parsed sentences are now becoming available.

BOX 7.1 Getting Reliable Statistics

Given that you need to estimate probabilities for lexical items and for n-grams, how much data is needed for these estimates to be reliable? In practice, the amount of data needed depends heavily on the size of the n-grams used, as the number of probabilities that need to be estimated grows rapidly with n. For example, a typical tagset has about 40 different lexical categories. To collect statistics on a unigram (a simple count of the words in each category), you would only need 40 statistics, one for each category. For bigrams, you would need 1600 statistics, one for each pair. For trigrams, you would need 64,000, one for each possible triple. Finally, for four-grams, you would need 2,560,000 statistics. As you can see, even if the corpus is a million words, a four-gram analysis would result in most categories being empty. For trigrams and a million-word corpus, however, there would be an average of 15 examples per category if they were evenly distributed. While the trigrams are definitely not uniformly distributed, this amount of data seems to give good results in practice.

One technique that is very useful for handling sparse data is called **smoothing**. Rather than simply using a trigram to estimate the probability of a category C_i, at position i, you use a formula that combines the trigram, bigram, and unigram statistics. Using this scheme, the probability of category C_i given the preceding categories $C_1, ..., C_{i-1}$ is estimated by the formula

$$PROB(C_i \mid C_1, ..., C_{i-1}) \cong \lambda_1 PROB(C_i) + \lambda_2 PROB(C_i \mid C_{i-1}) + \lambda_3 PROB(C_i \mid C_{i-2}\ C_{i-1})$$

where $\lambda_1+\lambda_2+\lambda_3 = 1$. Using this estimate, if the trigram has never been seen before, the bigram or unigram estimates still would guarantee a nonzero estimate in many cases where it is desired. Typically, the best performance will arise if λ_3 is significantly greater than the other parameters so that the trigram information has the most effect on the probabilities. It is also possible to develop algorithms that learn good values for the parameters given a particular training set (for example, see Jelinek (1990)).

The first issue is what the input would be to such a parser. One simple approach would be to use a part-of-speech tagging algorithm from the last section to select a single category for each word and then start the parse with these categories. If the part-of-speech tagging is accurate, this will be an excellent approach, because a considerable amount of lexical ambiguity will be eliminated before the parser even starts. But if the tagging is wrong, it will prevent the parser from ever finding the correct interpretation. Worse, the parser may find a valid but implausible interpretation based on the wrongly tagged word and never realize the error. Consider that even at 95 percent accuracy, the chance that every word is correct in a sentence consisting of only 8 words is .67, and with 12 words it is .46—less than half. Thus the chances of this approach working in general look slim.

PROB(ART | *the*) ≅	.99	*PROB*(N | *like*) ≅	.16
PROB(N | *flies*) ≅	.48	*PROB*(ART | *a*) ≅	.995
PROB(V | *flies*) ≅	.52	*PROB*(N | *a*) ≅	.005
PROB(V | *like*) ≅	.49	*PROB*(N | *flower*) ≅	.78
PROB(P | *like*) ≅	.34	*PROB*(V | *flower*) ≅	.22

Figure 7.13 Context-independent estimates for the lexical categories

A more appropriate approach would be to compute the probability that each word appears in the possible lexical categories. If we could combine these probabilities with some method of assigning probabilities to rule use in the grammar, then we could develop a parsing algorithm that finds the most probable parse for a given sentence.

You already saw the simplest technique for estimating lexical probability by counting the number of times each word appears in the corpus in each category. Then the probability that word w appears in a lexical category L_j out of possible categories $L_1, ..., L_N$ could be estimated by the formula

$$PROB(L_j \mid w) \cong \text{count}(L_j \,\&\, w) / \Sigma_{i=1,N}\, \text{count}(L_i \,\&\, w)$$

Using the data shown in Figure 7.5, we could derive the context-independent probabilities for each category and word shown in Figure 7.13.

As we saw earlier, however, such estimates are unreliable because they do not take context into account. A better estimate would be obtained by computing how likely it is that category L_i occurred at position t over all sequences given the input $w_1, ..., w_t$. In other words, rather than searching for the one sequence that yields the maximum probability for the input, we want to compute the sum of the probabilities for the input from all sequences.

For example, the probability that *flies* is a noun in the sentence *The flies like flowers* would be calculated by summing the probability of all sequences that end with *flies* as a noun. Given the transition and lexical-generation probabilities in Figures 7.4 and 7.6, the sequences that have a nonzero values would be

The/ART flies/N	$9.58 * 10^{-3}$
The/N flies/N	$1.13 * 10^{-6}$
The/P flies/N	$4.55 * 10^{-9}$

which adds up to $9.58 * 10^{-3}$. Likewise, three nonzero sequences end with *flies* as a V, yielding a total sum of $1.13 * 10^{-5}$. Since these are the only sequences that have nonzero scores when the second word is *flies*, the sum of all these sequences will be the probability of the sequence *The flies*, namely $9.591 * 10^{-3}$. We can now compute the probability that *flies* is a noun as follows:

$$PROB(\text{flies/N} \mid \textit{The flies})$$
$$= PROB(\text{flies/N} \,\&\, \textit{The flies}) \,/\, PROB(\textit{The flies})$$

Initialization Step

For i = 1 to N do
    SEQSUM(i, 1) = $PROB(w_1 \mid L_i) * PROB(L_i \mid \emptyset)$

Computing the Forward Probabilities

For t = 2 to T do
    For i = 1 to N do
        SEQSUM(i, t) = $\Sigma_{j=1,N}$ ($PROB(L_i \mid L_j)$ * SEQSUM(j, t–1)) * $PROB(w_t \mid L_i)$

Computing the Lexical Probabilities

For t = 1 to T do
    For i = 1 to N do
        $PROB(C_t = L_i)$ = SEQSUM(i, t) / $\Sigma_{j=1,N}$ SEQSUM(j, t)

Figure 7.14 The forward algorithm for computing the lexical probabilities

$$= 9.58 * 10^{-3} / 9.591 * 10^{-3}$$
$$= .9988$$

Likewise, the probability that *flies* is a verb would be .0012.

Of course, it would not be feasible to enumerate all possible sequences in a realistic example. Luckily, however, the same trick used in the Viterbi algorithm can be used here. Rather than selecting the maximum score for each node at each stage of the algorithm, we compute the sum of all scores.

To develop this more precisely, we define the **forward probability**, written as $\alpha_i(t)$, which is the probability of producing the words $w_1, ..., w_t$ and ending in state w_t/L_i:

$$\alpha_i(t) = PROB(w_t/L_i, w_1, ..., w_t)$$

For example, with the sentence *The flies like flowers*, $\alpha_2(3)$ would be the sum of values computed for all sequences ending in V (the second category) in position 3 given the input *The flies like*. Using the definition of conditional probability, you can then derive the probability that word w_t is an instance of lexical category L_i as follows:

$$PROB(w_t/L_i \mid w_1, ..., w_t) = PROB(w_t/L_i, w_1, ..., w_t) / PROB(w_1, ..., w_t)$$

We estimate the value of $PROB(w_1, ..., w_t)$ by summing over all possible sequences up to any state at position t, which is simply $\Sigma_{j=1,N} \alpha_j(t)$. In other words, we end up with

$$PROB(w_t/L_i \mid w_1, ..., w_t) \cong \alpha_i(t) / \Sigma_{j=1,N} \alpha_j(t)$$

The first two parts of the algorithm shown in Figure 7.14 compute the forward probabilities using a variant of the Viterbi algorithm. The last step converts the

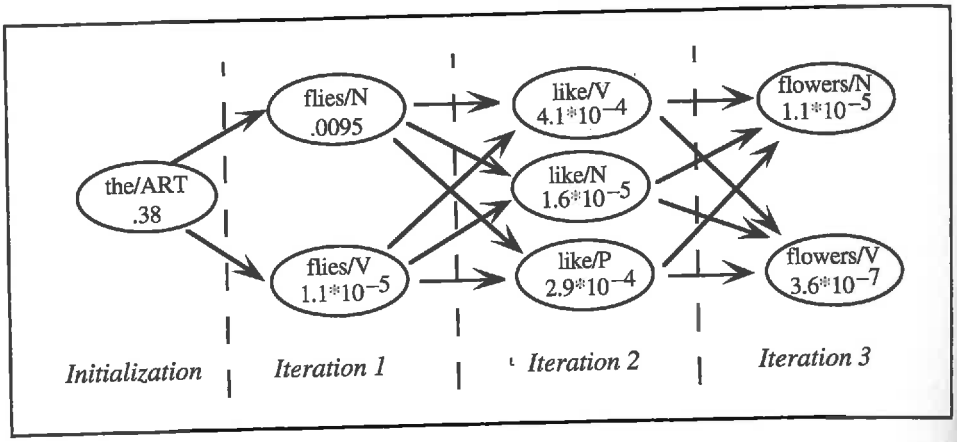

Figure 7.15 Computing the sums of the probabilities of the sequences

PROB(the/ART | *the*) =	1.0	*PROB*(like/P | *the flies like*) ≅	.4
PROB(flies/N | *the flies*) ≅	.9988	*PROB*(like/N | *the flies like*) ≅	.022
PROB(flies/V | *the flies*) ≅	.0011	*PROB*(flowers/N | *the flies like flowers*) ≅	.967
PROB(like/V | *the flies like*) ≅	.575	*PROB*(flowers/V | *the flies like flowers*) ≅	.033

Figure 7.16 Context-dependent estimates for lexical categories in the sentence *The flies like flowers*

forward probabilities into lexical probabilities for the given sentence by normalizing the values.

Consider deriving the lexical probabilities for the sentence *The flies like flowers* using the probability estimates in Figures 7.4 and 7.6. The algorithm in Figure 7.14 would produce the sums shown in Figure 7.15 for each category in each position, resulting in the probability estimates shown in Figure 7.16.

Note that while the context-independent approximation in Figure 7.13 slightly favors the verb interpretation of *flies*, the context-dependent approximation virtually eliminates it because the training corpus had no sentences with a verb immediately following an article. These probabilities are significantly different than the context-independent ones and much more in line with intuition.

Note that you could also consider the **backward probability**, $\beta_i(t)$, the probability of producing the sequence $w_t, ..., w_T$ beginning from state w_t/L_i. These values can be computed by an algorithm similar to the forward probability algorithm but starting at the end of the sentence and sweeping backward through the states. Thus a better method of estimating the lexical probabilities for word w_t would be to consider the entire sentence rather than just the words up to t. In this case, the estimate would be

$$PROB(w_t/L_i) = (\alpha_i(t) * \beta_i(t)) / \Sigma_{j=1,N} (\alpha_j(t) * \beta_j(t))$$