9 *Markov Models*

HIDDEN MARKOV MODELS (HMMs) have been the mainstay of the statistical modeling used in modern speech recognition systems. Despite their limitations, variants of HMMs are still the most widely used technique in that domain, and are generally regarded as the most successful. In this chapter we will develop the basic theory of HMMs, touch on their applications, and conclude with some pointers on extending the basic HMM model and engineering practical implementations.

HIDDEN MARKOV MODEL

An *HMM* is nothing more than a probabilistic function of a Markov process. We have already seen an example of Markov processes in the n-gram models of Chapters 2 and 6. *Markov processes/chains/models* were first developed by Andrei A. Markov (a student of Chebyshev). Their first use was actually for a linguistic purpose – modeling the letter sequences in works of Russian literature (Markov 1913) – but Markov models were then developed as a general statistical tool. We will refer to vanilla Markov models as Visible Markov Models (VMMs) when we want to be careful to distinguish them from HMMs.

MARKOV MODEL

We have placed this chapter at the beginning of the "grammar" part of the book because working on the order of words in sentences is a start at understanding their syntax. We will see that this is what a VMM does. HMMs operate at a higher level of abstraction by postulating additional "hidden" structure, and that allows us to look at the order of *categories* of words. After developing the theory of HMMs in this chapter, we look at the application of HMMs to part-of-speech tagging. The last two chapters in this part then deal with the probabilistic formalization of core notions of grammar like phrase structure.

9.1 Markov Models

Often we want to consider a sequence (perhaps through time) of random variables that *aren't* independent, but rather the value of each variable depends on previous elements in the sequence. For many such systems, it seems reasonable to assume that all we need to predict the future random variables is the value of the present random variable, and we don't need to know the values of all the past random variables in the sequence. For example, if the random variables measure the number of books in the university library, then, knowing how many books were in the library today might be an adequate predictor of how many books there will be tomorrow, and we don't really need to additionally know how many books the library had last week, let alone last year. That is, future elements of the sequence are conditionally independent of past elements, given the present element.

MARKOV ASSUMPTION

Suppose $X = (X_1, \ldots, X_T)$ is a sequence of random variables taking values in some finite set S = $\{s_1, \ldots, s_N\}$, the state space. Then the *Markov Properties* are:

Limited Horizon:

$$P(X_{t+1} = s_k | X_1, \ldots, X_t) = P(X_{t+1} = s_k | X_t) \tag{9.1}$$

Time invariant (stationary):

$$= P(X_2 = s_k | X_1) \tag{9.2}$$

X is then said to be a Markov chain, or to have the Markov property. One can describe a Markov chain by a stochastic transition matrix A:

$$a_{ij} = P(X_{t+1} = s_j | X_t = s_i) \tag{9.3}$$

Here, $a_{ij} \geq 0, \forall i, j$ and $\sum_{j=1}^{N} a_{ij} = 1, \forall i$.

Additionally one needs to specify Π, the probabilities of different initial states for the Markov chain:

$$\pi_i = P(X_1 = s_i) \tag{9.4}$$

Here, $\sum_{i=1}^{N} \pi_i = 1$. The need for this vector can be avoided by specifying that the Markov model always starts off in a certain extra initial state, s_0, and then using transitions from that state contained within the matrix A to specify the probabilities that used to be recorded in Π.

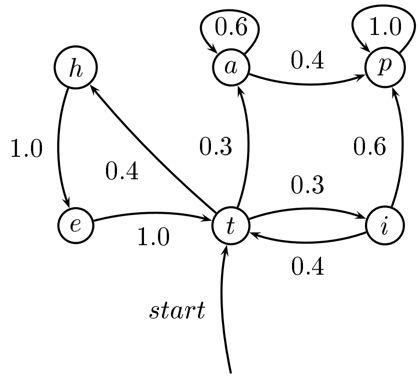

Figure 9.1 A Markov model.

From this general description, it should be clear that the word n-gram models we saw in Chapter 6 are Markov models. Markov models can be used whenever one wants to model the probability of a linear sequence of events. For example, they have also been used in NLP for modeling valid phone sequences in speech recognition, and for sequences of speech acts in dialog systems.

Alternatively, one can represent a Markov chain by a state diagram as in Figure 9.1. Here, the states are shown as circles around the state name, and the single start state is indicated with an incoming arrow. Possible transitions are shown by arrows connecting states, and these arcs are labeled with the probability of this transition being followed, given that you are in the state at the tail of the arrow. Transitions with zero probability are omitted from the diagram. Note that the probabilities of the outgoing arcs from each state sum to 1. From this representation, it should be clear that a Markov model can be thought of as a (nondeterministic) finite state automaton with probabilities attached to each arc. The Markov properties ensure that we have a finite state automaton. There are no long distance dependencies, and where one ends up next depends simply on what state one is in.

In a visible Markov model, we know what states the machine is passing through, so the state sequence or some deterministic function of it can be regarded as the output.

The probability of a sequence of states (that is, a sequence of random variables) $X_1, \ldots, X_T$ is easily calculated for a Markov chain. We find that we need merely calculate the product of the probabilities that occur on the arcs or in the stochastic matrix:

$$\begin{aligned} P(X_1, \ldots, X_T) &= P(X_1)P(X_2|X_1)P(X_3|X_1, X_2) \cdots P(X_T|X_1, \ldots, X_{T-1}) \\ &= P(X_1)P(X_2|X_1)P(X_3|X_2) \cdots P(X_T|X_{T-1}) \\ &= \pi_{X_1} \prod_{t=1}^{T-1} a_{X_t X_{t+1}} \end{aligned}$$

So, using the Markov model in Figure 9.1, we have:

$$\begin{aligned} P(t, i, p) &= P(X_1 = t)P(X_2 = i|X_1 = t)P(X_3 = p|X_2 = i) \\ &= 1.0 \times 0.3 \times 0.6 \\ &= 0.18 \end{aligned}$$

Note that what is important is whether we *can* encode a process as a Markov process, not whether we most naturally do. For example, recall the n-gram word models that we saw in Chapter 6. One might think that, for $n \geq 3$, such a model is not a Markov model because it violates the Limited Horizon condition – we are looking a little into earlier history. But we can reformulate any n-gram model as a visible Markov model by simply encoding the appropriate amount of history into the state space (states are then $(n-1)$-grams, for example (*was*, *walking*, *down*) would be a state in a fourgram model). In general, any fixed finite amount of history can always be encoded in this way by simply elaborating the state space as a crossproduct of multiple previous states. In such cases, we sometimes talk of an m^{th} *order* Markov model, where m is the number of previous states that we are using to predict the next state. Note, thus, that an n-gram model is equivalent to an $(n-1)^{\text{th}}$ order Markov model.

Exercise 9-1
Build a Markov Model similar to Figure 9.1 for one of the types of phone numbers in Table 4.2.

9.2 Hidden Markov Models

In an HMM, you don't know the state sequence that the model passes through, but only some probabilistic function of it.

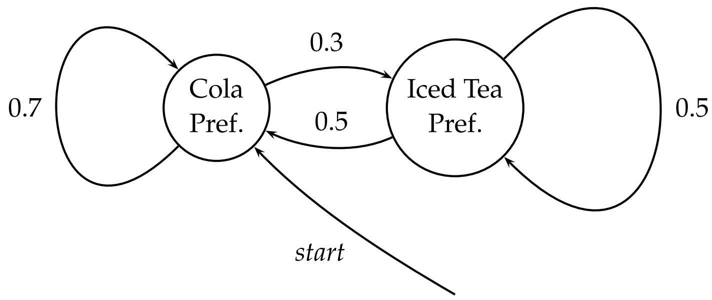

Figure 9.2 The crazy soft drink machine, showing the states of the machine and the state transition probabilities.

Example 1: Suppose you have a crazy soft drink machine: it can be in two states, cola preferring (CP) and iced tea preferring (IP), but it switches between them randomly after each purchase, as shown in Figure 1.

Now, if, when you put in your coin, the machine always put out a cola if it was in the cola preferring state and an iced tea when it was in the iced tea preferring state, then we would have a visible Markov model. But instead, it only has a tendency to do this. So we need symbol emission probabilities for the observations:

$P(O_t = k|X_t = s_i, X_{t+1} = s_j) = b_{ijk}$

For this machine, the output is actually independent of s_j, and so can be described by the following probability matrix:

Output probability given From state

	cola	iced tea ice_t	lemonade lem
CP	0.6	0.1	0.3
IP	0.1	0.7	0.2

What is the probability of seeing the output sequence {lem, ice_t} if the machine always starts off in the cola preferring state?

Solution: We need to consider all paths that might be taken through the HMM, and then to sum over them. We know the machine starts in state CP. There are then four possibilities depending on which of the two states the machine is in at the other two time instants. So the total probability is:

$$\begin{aligned} 0.7 \times 0.3 \times 0.7 \times 0.1 + 0.7 \times 0.3 \times 0.3 \times 0.1 &\quad + \\ 0.3 \times 0.3 \times 0.5 \times 0.7 + 0.3 \times 0.3 \times 0.5 \times 0.7 &= \quad 0.084 \end{aligned}$$

Exercise 9-2
What is the probability of seeing the output sequence {col,lem } if the machine always starts off in the ice tea preferring state?

9.2.1 Why use HMMs?

HMMs are useful when one can think of underlying events probabilistically generating surface events. One widespread use of this is tagging – assigning parts of speech (or other classifiers) to the words in a text. We think of there being an underlying Markov chain of parts of speech from which the actual words of the text are generated. Such models are discussed in Chapter 10.

When this general model is suitable, the further reason that HMMs are very useful is that they are one of a class of models for which there exist efficient methods of training through use of the Expectation Maximization (EM) algorithm. Given plenty of data that we assume to be generated by *some* HMM – where the model architecture is fixed but not the probabilities on the arcs – this algorithm allows us to automatically learn the model parameters that best account for the observed data.

Another simple illustration of how we can use HMMs is in generating parameters for linear interpolation of n-gram models. We discussed in Chapter 6 that one way to estimate the probability of a sentence:

$$P(\text{Sue drank her beer before the meal arrived})$$

was with an n-gram model, such as a trigram model, but that just using an n-gram model with fixed n tended to suffer because of data sparseness. Recall from Section 6.3.1 that one idea of how to smooth n-gram estimates was to use linear interpolation of n-gram estimates for various n, for example:

$$P_{\text{li}}(w_n|w_{n-1}, w_{n-2}) = \lambda_1 P_1(w_n) + \lambda_2 P_2(w_n|w_{n-1}) + \lambda_3 P_3(w_n|w_{n-1}, w_{n-2})$$

This way we would get some idea of how likely a particular word was, even if our coverage of trigrams is sparse. The question, then, is how to set the parameters λ_i. While we could make reasonable guesses as to what parameter values to use (and we know that together they must obey the stochastic constraint $\sum_i \lambda_i = 1$), it seems that we should be able to find the optimal values automatically. And, indeed, we can (Jelinek 1990).

The key insight is that we can build an HMM with hidden states that represent the choice of whether to use the unigram, bigram, or trigram probabilities. The HMM training algorithm will determine the optimal weight

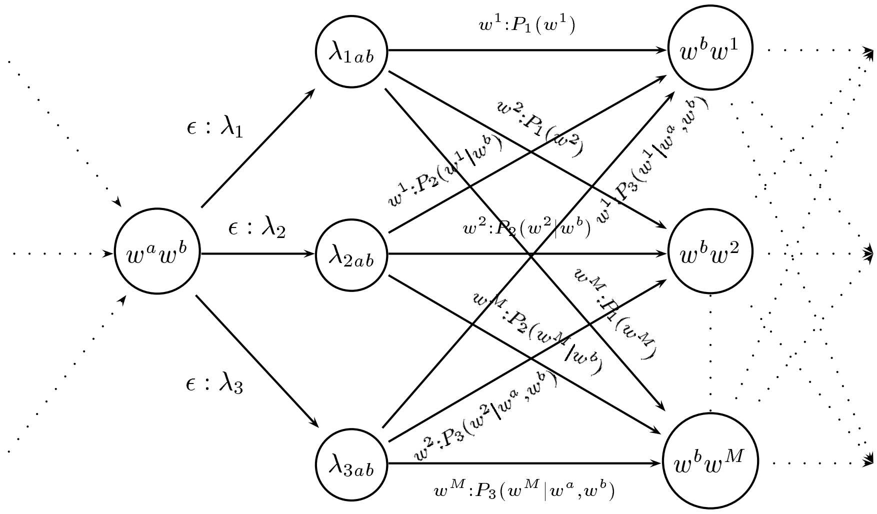

Figure 9.3 A section of an HMM for a linearly interpolated language model. The notation $o : p$ on arcs means that this transition is made with probability p, and that an o is output when this transition is made (with probability 1).

to give to the arcs entering each of these hidden states, which in turn represents the amount of the probability mass that should be determined by each n-gram model via setting the parameters λ_i above.

Concretely, we build an HMM with four states for each word pair, one for the basic word pair, and three representing each choice of n-gram model for calculating the next transition. A fragment of the HMM is shown in Figure 9.3. Note how this HMM assigns the same probabilities as the earlier equation: there are three ways for w^c to follow $w^a w^b$ and the total probability of seeing w^c next is then the sum of each of the n-gram probabilities that adorn the arcs multiplied by the corresponding parameter λ_i. The HMM training algorithm that we develop in this chapter can then be applied to this network, and used to improve initial estimates for the parameters λ_{iab}.

EPSILON TRANSITIONS

There are two things to note. This conversion works by adding *epsilon transitions* – that is transitions that we wish to say do not produce an output symbol. Secondly, as presented, we now have separate parameters λ_{iab} for each word pair. But we would not want to adjust these parameters separately, as this would make our sparse data problem worse not better. Rather, for a fixed i, we wish to keep all (or at least classes of) the λ_{iab} pa-

Set of states	$S = \{s_1, \ldots s_N\}$
Output alphabet	$K = \{k_1, \ldots, k_M\} = \{1, \ldots, M\}$
Intial state probabilities	$\Pi = \{\pi_i\}, i \in S$
State transition probabilities	$A = \{a_{ij}\}, i, j \in S$
Symbol emission probabilities	$B = \{b_{ijk}\}, i, j \in S, k \in K$
State sequence	$X = (X_1, \ldots, X_{T+1}) \quad X_t : S \mapsto \{1, \ldots, N\}$
Output sequence	$O = (o_1, \ldots, o_T) \quad o_t \in K$

Table 9.1 Notation used in the HMM chapter.

rameters having the same value, which we do by using *tied states*. Discussion of both of these extensions to the basic HMM model will be deferred to Section 9.4.

9.2.2 General form of an HMM

An HMM is specified by a five-tuple (S, K, Π, A, B), where S and K are the set of states and the output alphabet, and Π, A, and B are the probabilities for the initial state, state transitions, and symbol emissions, respectively. The notation that we use in this chapter is summarized in Table 9.1. The random variables X_t map from state names to corresponding integers. In the version presented here, the symbol emitted at time t depends on both the state at time t and at time $t+1$. This is sometimes called a *arc-emission HMM*, because we can think of the symbol as coming off the arc, as in Figure 9.3. An alternative formulation is a *state-emission HMM*, where the symbol emitted at time t depends just on the state at time t. The HMM in Example 1 is a state-emission HMM. But we can also regard it as a arc-emission HMM by simply setting up the b_{ijk} parameters so that $\forall k', k''$, $b_{ijk'} = b_{ijk''}$. This is discussed further in Section 9.4.

ARC-EMISSION HMM

STATE-EMISSION HMM

Given a specification of an HMM, it is perfectly straightforward to simulate the running of a Markov process, and to produce an output sequence. One can do it with the program in Figure 9.4. However, by itself, doing this is not terribly interesting. The interest in HMMs comes from *assuming* that some set of data was generated by a HMM, and then being able to calculate probabilities and probable underlying state sequences.

```
t := 1;
Start in state s_i with probability π_i (i.e., X1 = i)
forever do
        Move from state s_i to state s_j with probabilitiy a_ij (i.e., X_{t+1} = j)
        Emit observation symbol o_t = k with probability b_ijk
        t := t + 1
od
```

Figure 9.4 A program for a Markov process.

9.3 The Three Fundamental Questions for HMMs

There are three fundamental questions that we want to know about an HMM:

1. Given a model $\mu = (A, B, \Pi)$, how do we efficiently compute how likely a certain observation is, that is $P(O|\mu)$?

2. Given the observation sequence O and a model μ, how do we choose a state sequence $(X_1, \ldots, X_{T+1})$ that best explains the observations?

3. Given an observation sequence O, and a space of possible models found by varying the model parameters $\mu = (A, B, \pi)$, how do we find the model that best explains the observed data?

Normally, the problems we deal with are not like the soft drink machine. We don't know the parameters and have to estimate them from data. That's the third question. The first question can be used to decide between models which is best. The second question lets us guess what path was probably followed through the Markov chain, and this hidden path can be used for classification, for instance in applications to part of speech tagging, as we see in Chapter 10.

9.3.1 Finding the probability of an observation

Given the observation sequence $O = (o_1, \ldots, o_T)$ and a model $\mu = (A, B, \Pi)$, we wish to know how to efficiently compute $P(O|\mu)$ – the probability of the observation given the model. This process is often referred to as "decoding".

For any state sequence $X = (X_1, \ldots, X_{T+1})$,

(9.5)
$$P(O|X,\mu) = \prod_{t=1}^{T} P(o_t|X_t, X_{t+1}, \mu) = b_{X_1X_2o_1}b_{X_2X_3o_2}\cdots b_{X_TX_{T+1}o_T}$$

and,

(9.6)
$$P(X|\mu) = \pi_{X_1}a_{X_1X_2}a_{X_2X_3}\cdots a_{X_TX_{T+1}}$$

Now,

(9.7)
$$P(O, X|\mu) = P(O|X,\mu)P(X|\mu)$$

Therefore,

(9.8)
$$P(O|\mu) = \sum_X P(O|X,\mu)P(X|\mu) = \sum_{X_1\cdots X_{T+1}} \pi_{X_1}\prod_{t=1}^{T} a_{X_tX_{t+1}}b_{X_tX_{t+1}o_t}$$

This derivation is quite straightforward. It is what we did in Example 1 to work out the probability of an observation sequence. We simply summed the probability of the observation occurring according to each possible state sequence. But, unfortunately, direct evaluation of the resulting expression is hopelessly inefficient. For the general case (where one can start in any state, and move to any other at each step), the calculation requires $(2T+1)\cdot N^{T+1}$ multiplications.

Exercise 9-3
Confirm this claim.

DYNAMIC PROGRAMMING
MEMOIZATION

The secret to avoiding this complexity is the general technique of *dynamic programming* or *memoization* by which we remember partial results rather than recomputing them. This general concept crops up in many other places in computational linguistics, such as chart parsing, and in computer science more generally (see (Cormen et al. 1990: Ch. 16) for a general introduction). For algorithms such as HMMs, the dynamic programming problem is generally described in terms of trellises (also called lattices). Here, we make a square array of states versus time, and compute the probabilities of being at each state at each time in terms of the probabilities for being in each state at the preceding time instant. This is all best seen in pictures – see Figures 9.5 and 9.6. A trellis can record the probability of all initial

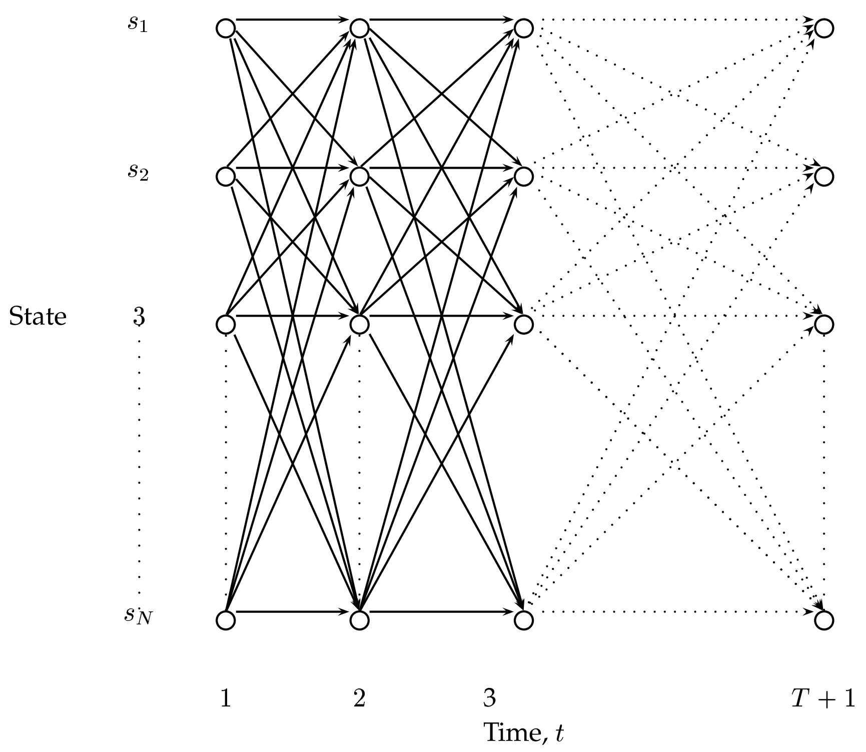

Figure 9.5 Trellis algorithms: The trellis. The trellis is a square array of states versus times. A node at (s_i, t) can store information about state sequences which include $X_t = i$. The lines show the connections between nodes. Here we have a fully interconnected HMM where one can move from any state to any other at each step.

subpaths of the HMM that end in a certain state at a certain time. The probability of longer subpaths can then be worked out in terms of one shorter subpaths.

The forward procedure

FORWARD PROCEDURE

The form of caching that is indicated in these diagrams is called the *forward procedure*. We describe it in terms of forward variables:

$$\alpha_i(t) = P(o_1 o_2 \cdots o_{t-1}, X_t = i|\mu) \tag{9.9}$$

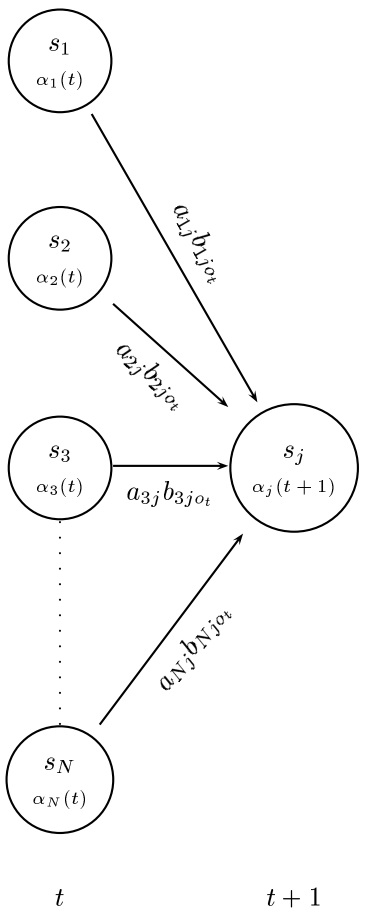

Figure 9.6 Trellis algorithms: Closeup of the computation of forward probabilities at one node. The forward probability $\alpha_j(t+1)$ is calculated by summing the product of the probabilities on each incoming arc with the forward probability of the originating node.

The forward variable $\alpha_i(t)$ is stored at (s_i, t) in the trellis and expresses the total probability of ending up in state s_i at time t (given that the observations $o_1 \cdot o_{t-1}$ were seen). It is calculated by summing probabilities for all incoming arcs at a trellis node. We calculate the forward variables in the trellis left to right using the following procedure:

1. Initialization

$$\alpha_i(1) = \pi_i, \quad 1 \le i \le N$$

2. Induction

$$\alpha_j(t+1) = \sum_{i=1}^{N} \alpha_i(t) a_{ij} b_{ijo_t}, \quad 1 \le t \le T, 1 \le j \le N$$

3. Total

$$P(O|\mu) = \sum_{i=1}^{N} \alpha_i(T+1)$$

This is a much cheaper algorithm that requires only $2N^2T$ multiplications.

The backward procedure

It should be obvious that we do not need to cache results working forward through time like this, but rather that we could also work backward. The BACKWARD PROCEDURE *backward procedure* computes backward variables which are the total probability of seeing the rest of the observation sequence given that we were in state s_i at time t. The real reason for introducing this less intuitive calculation, though, is because use of a combination of forward and backward probabilities is vital for solving the third problem of parameter reestimation.

Define backward variables

(9.10) $$\beta_i(t) = P(o_t \cdots o_T | X_t = i, \mu)$$

Then we can calculate backward variables working from right to left through the trellis as follows:

1. Initialization

$$\beta_i(T+1) = 1, \quad 1 \le i \le N$$

2. Induction

$$\beta_i(t) = \sum_{j=1}^{N} a_{ij} b_{ijo_t} \beta_j(t+1), \quad 1 \le t \le T, 1 \le i \le N$$

	Output			
	lem	ice_t	cola	
Time (t):	1	2	3	4
$\alpha_{CP}(t)$	1.0	0.21	0.0462	0.021294
$\alpha_{IP}(t)$	0.0	0.09	0.0378	0.010206
$P(o_1 \cdots o_{t-1})$	1.0	0.3	0.084	0.0315
$\beta_{CP}(t)$	0.0315	0.045	0.6	1.0
$\beta_{IP}(t)$	0.029	0.245	0.1	1.0
$P(o_1 \cdots o_T)$	0.0315			
$\gamma_{CP}(t)$	1.0	0.3	0.88	0.676
$\gamma_{IP}(t)$	0.0	0.7	0.12	0.324
$\widehat{X_t}$	CP	IP	CP	CP
$\delta_{CP}(t)$	1.0	0.21	0.0315	0.019404
$\delta_{IP}(t)$	0.0	0.09	0.0315	0.008316
$\psi_{CP}(t)$		CP	IP	CP
$\psi_{IP}(t)$		CP	IP	CP
$\hat{X}_t$	CP	IP	CP	CP
$P(\hat{X})$	0.019404			

Table 9.2 Variable calculations for $O =$ (lem, ice_t, cola).

3. Total

$$P(O|\mu) = \sum_{i=1}^{N} \pi_i \beta_i(1)$$

Table 9.2 shows the calculation of forward and backward variables, and certain other variables that we will come to later for the soft drink machine from Example 1, given the observation sequence $O =$ (lem, ice_t, cola).

Combining them

More generally, in fact, we can use any combination of forward and backward caching to work out the probability of an observation sequence. Observe that:

$$\begin{aligned} P(O, X_t = i|\mu) &= P(o_1 \cdots o_T, X_t = i|\mu) \\ &= P(o_1 \cdots o_{t-1}, X_t = i, o_t \cdots o_T|\mu) \end{aligned}$$

$$
\begin{aligned}
&= P(o_1 \cdots o_{t-1}, X_t = i|\mu) P(o_t \cdots o_T | o_1 \cdots o_{t-1}, X_t = i, \mu) \\
&= P(o_1 \cdots o_{t-1}, X_t = i|\mu) P(o_t \cdots o_T | X_t = i, \mu) \\
&= \alpha_i(t)\beta_i(t)
\end{aligned}
$$

Therefore:

(9.11) $$P(O|\mu) = \sum_{i=1}^{N} \alpha_i(t)\beta_i(t), \quad 1 \le t \le T+1$$

The previous equations were special cases of this one.

9.3.2 Finding the best state sequence

The second problem was worded somewhat vaguely as "finding the state sequence that best explains the observations". That is because there is more than one way to think about doing this. One way to proceed would be to choose the states individually. That is, for each t, $1 \le t \le T+1$, we would find X_t that maximizes $P(X_t|O,\mu)$.

Let

(9.12) $$
\begin{aligned}
\gamma_i(t) &= P(X_t = i|O,\mu) \\
&= \frac{P(X_t = i, O|\mu)}{P(O|\mu)} \\
&= \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}
\end{aligned}
$$

The individually most likely state $\widehat{X_t}$ is:

(9.13) $$\widehat{X_t} = \operatorname*{arg\,max}_{1 \le i \le N} \gamma_i(t), \quad 1 \le t \le T+1$$

This quantity maximizes the expected number of states that will be guessed correctly. However, it may yield a quite unlikely state *sequence*. Therefore, this is not the method that is normally used, but rather the Viterbi algorithm, which efficiently computes the most likely state sequence.

Viterbi algorithm

Commonly we want to find the most likely complete path, that is:

$$\operatorname*{arg\,max}_{X} P(X|O,\mu)$$

To do this, it is sufficient to maximize for a fixed O:

$$\arg\max_X P(X, O|\mu)$$

VITERBI ALGORITHM

An efficient trellis algorithm for computing this path is the *Viterbi algorithm*. Define:

$$\delta_j(t) = \max_{X_1 \cdots X_{t-1}} P(X_1 \cdots X_{t-1}, o_1 \cdots o_{t-1}, X_t = j|\mu)$$

This variable stores for each point in the trellis the probability of the most probable path that leads to that node. The corresponding variable $\psi_j(t)$ then records the node of the incoming arc that led to this most probable path. Using dynamic programming, we calculate the most probable path through the whole trellis as follows:

1. Initialization

   $$\delta_j(1) = \pi_j, \quad 1 \leq j \leq N$$

2. Induction

   $$\delta_j(t+1) = \max_{1 \leq i \leq N} \delta_i(t) a_{ij} b_{ijo_t}, \quad 1 \leq j \leq N$$

   Store backtrace

   $$\psi_j(t+1) = \arg\max_{1 \leq i \leq N} \delta_i(t) a_{ij} b_{ijo_t}, 1 \leq j \leq N$$

3. Termination and path readout (by backtracking). The most likely state sequence is worked out from the right backwards:

   $$\begin{aligned} \hat{X}_{T+1} &= \arg\max_{1 \leq i \leq N} \delta_i(T+1) \\ \hat{X}_t &= \psi_{\hat{X}_{t+1}}(t+1) \\ P(\hat{X}) &= \max_{1 \leq i \leq N} \delta_i(T+1) \end{aligned}$$

In these calculations, one may get ties. We assume that in that case one path is chosen randomly. In practical applications, people commonly want to work out not only the best state sequence but the n-best sequences or a graph of likely paths. In order to do this people often store the $m < n$ best previous states at a node.

Table 9.2 above shows the computation of the most likely states and state sequence under both these interpretations – for this example, they prove to be identical.

9.3.3 The third problem: Parameter estimation

Given a certain observation sequence, we want to find the values of the model parameters $\mu = (A, B, \pi)$ which best explain what we observed. Using Maximum Likelihood Estimation, that means we want to find the values that maximize $P(O|\mu)$:

(9.14) $$\arg\max_{\mu} P(O_{\text{training}}|\mu)$$

There is no known analytic method to choose μ to maximize $P(O|\mu)$. But we can locally maximize it by an iterative hill-climbing algorithm. This algorithm is the Baum-Welch or Forward-Backward algorithm, which is a special case of the Expectation Maximization method which we covered in greater generality in Section 14.2.2. It works like this. We don't know what the model is, but we can work out the probability of the observation sequence using some (perhaps randomly chosen) model. Looking at that calculation, we can see which state transitions and symbol emissions were probably used the most. By increasing the probability of those, we can choose a revised model which gives a higher probability to the observation sequence. This maximization process is often referred to as *training* the model and is performed on *training data*.

TRAINING

TRAINING DATA

Define $p_t(i,j), 1 \le t \le T, 1 \le i,j \le N$ as shown below. This is the probability of traversing a certain arc at time t given observation sequence O; see Figure 9.7.

(9.15)
$$\begin{aligned}
p_t(i,j) &= P(X_t = i, X_{t+1} = j|O, \mu) \\
&= \frac{P(X_t = i, X_{t+1} = j, O|\mu)}{P(O|\mu)} \\
&= \frac{\alpha_i(t) a_{ij} b_{ijo_t} \beta_j(t+1)}{\sum_{m=1}^{N} \alpha_m(t)\beta_m(t)} \\
&= \frac{\alpha_i(t) a_{ij} b_{ijo_t} \beta_j(t+1)}{\sum_{m=1}^{N} \sum_{n=1}^{N} \alpha_m(t) a_{mn} b_{mno_t} \beta_n(t+1)}
\end{aligned}$$

Note that $\gamma_i(t) = \sum_{j=1}^{N} p_t(i,j)$.

Now, if we sum over the time index, this gives us expectations (counts):

$$\sum_{t=1}^{T} \gamma_i(t) = \text{expected number of transitions from state } i \text{ in } O$$

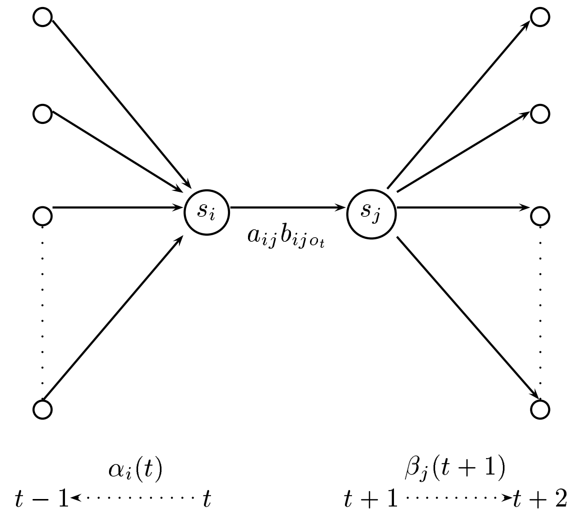

Figure 9.7 The probability of traversing an arc. Given an observation sequence and a model, we can work out the probability that the Markov proess went from state s_i to s_j at time t.

$$\sum_{t=1}^{T} p_t(i,j) = \text{expected number of transitions from state } i \text{ to } j \text{ in } O$$

So we begin with some model μ (perhaps preselected, perhaps just chosen randomly). We then run O through the current model to estimate the expectations of each model parameter. We then change the model to maximize the values of the paths that are used a lot (while still respecting the stochastic constraints). We then repeat this process, hoping to converge on optimal values for the model parameters μ.

The reestimation formulas are as follows:

(9.16)
$$\begin{aligned}\hat{\pi}_i &= \text{expected frequency in state } i \text{ at time } t=1\\ &= \gamma_i(1)\end{aligned}$$

(9.17)
$$\begin{aligned}\hat{a}_{ij} &= \frac{\text{expected number of transitions from state } i \text{ to } j}{\text{expected number of transitions from state } i}\\ &= \frac{\sum_{t=1}^{T} p_t(i,j)}{\sum_{t=1}^{T} \gamma_i(t)}\end{aligned}$$

(9.18) $$\hat{b}_{ijk} = \frac{\text{expected number of transitions from } i \text{ to } j \text{ when } k \text{ is observed}}{\text{expected number of transitions from } i \text{ to } j} = \frac{\sum_{\{t: o_t = k, 1 \le t \le T\}} p_t(i,j)}{\sum_{t=1}^{T} p_t(i,j)}$$

Thus, from $\mu = (A, B, \Pi)$, we derive $\hat{\mu} = (\hat{A}, \hat{B}, \hat{\Pi})$. Further, as proved by Baum, we have that:

$$P(O|\hat{\mu}) \ge P(O|\mu)$$

This is a general property of the EM algorithm (see Section 14.2.2). Therefore, iterating through a number of rounds of parameter reestimation will improve our model. Normally one continues reestimating the parameters until results are no longer improving significantly. This process of parameter reestimation does not guarantee that we will find the best model, however, because the reestimation process may get stuck in a local maximum (or even possibly just at a saddle point). In most problems of interest, the likelihood function is a complex nonlinear surface and there are many local maxima. Nevertheless, Baum-Welch reestimation is usually effective for HMMs.

To end this section, let us consider reestimating the parameters of the crazy soft drink machine HMM using the Baum-Welch algorithm. If we let the initial model be the model that we have been using so far, then training on the observation sequence (lem, ice_t, cola) will yield the following values for $p_t(i, j)$:

		Time (and j)								
		1			2			3		
		CP	IP	γ_1	CP	IP	γ_2	CP	IP	γ_3
i	CP	0.3	0.7	1.0	0.28	0.02	0.3	0.616	0.264	0.88
	IP	0.0	0.0	0.0	0.6	0.1	0.7	0.06	0.06	0.12

and so the parameters will be reestimated as follows:

		Original			Reestimated		
Π	CP	1.0			1.0		
	IP	0.0			0.0		
		CP	IP		CP	IP	
A	CP	0.7	0.3		0.5486	0.4514	
	IP	0.5	0.5		0.8049	0.1951	
		cola	ice_t	lem	cola	ice_t	lem
B	CP	0.6	0.1	0.3	0.4037	0.1376	0.4587
	IP	0.1	0.7	0.2	0.1363	0.8537	0.0

Exercise 9-4

If one continued running the Baum-Welch algorithm on this HMM and this training sequence, what value would each parameter reach in the limit? Why?

The reason why the Baum-Welch algorithm is performing so strangely here should be apparent: the training sequence is far too short to accurately represent the behavior of the crazy soft drink machine.

Exercise 9-5

Note that the parameter that is zero in Π stays zero. Is that a chance occurrence? What would be the value of the parameter that becomes zero in B if we did another iteration of Baum-Welch reestimation? What generalization can one make about Baum-Welch reestimation of zero parameters?

9.4 HMMs: Implementation, Properties, and Variants

9.4.1 Implementation

Beyond the theory discussed above, there are a number of practical issues in the implementation of HMMs. Care has to be taken to make the implementation of HMM tagging efficient and accurate. The most obvious issue is that the probabilities we are calculating consist of keeping on multiplying together very small numbers. Such calculations will rapidly underflow the range of floating point numbers on a computer (even if you store them as 'double'!).

The Viterbi algorithm only involves multiplications and choosing the largest element. Thus we can perform the entire Viterbi algorithm working with logarithms. This not only solves the problem with floating point underflow, but it also speeds up the computation, since additions are much quicker than multiplications. In practice, a speedy implementation of the Viterbi algorithm is particularly important because this is the runtime algorithm, whereas training can usually proceed slowly offline.

However, in the Forward-Backward algorithm as well, something still has to be done to prevent floating point underflow. The need to perform summations makes it difficult to use logs. A common solution is to employ auxiliary scaling coefficients, whose values grow with the time t so that the probabilities multiplied by the scaling coefficient remain within the floating point range of the computer. At the end of each iteration, when the parameter values are reestimated, these scaling factors cancel out. Detailed discussion of this and other implementation issues can be found in (Levinson et al. 1983), (Rabiner and Juang 1993: 365–368), (Cutting et al. 1991), and (Dermatas and Kokkinakis 1995). The main alternative is to just use logs anyway, despite the fact that one needs to sum. Effectively then one is calculating an appropriate scaling factor at the time of each addition:

(9.19)
funct *log_add* $\equiv$
 if $(y - x > \log big)$
  then y
 elsif $(x - y > \log big)$
  then x
  else $\min(x, y) + \log(\exp(x - \min(x, y)) + \exp(y - \min(x, y)))$
 fi
.

where big is a suitable large constant like 10^{30}. For an algorithm like this where one is doing a large number of numerical computations, one also has to be careful about round-off errors, but such concerns are well outside the scope of this chapter.

9.4.2 Variants

There are many variant forms of HMMs that can be made without fundamentally changing them, just as with finite state machines. One is to allow some arc transitions to occur without emitting any symbol, so-called *epsilon* or *null transitions* (Bahl et al. 1983). Another commonly used variant is to make the output distribution dependent just on a single state, rather than on the two states at both ends of an arc as you traverse an arc, as was effectively the case with the soft drink machine. Under this model one can view the output as a function of the state chosen, rather than of the arc traversed. The model where outputs are a function of the state has actually been used more often in Statistical NLP, because it corresponds naturally to a part of speech tagging model, as we see in Chapter 10. Indeed,

EPSILON TRANSITIONS
NULL TRANSITIONS

some people will probably consider us perverse for having presented the arc-emission model in this chapter. But we chose the arc-emission model because it is trivial to simulate the state-emission model using it, whereas doing the reverse is much more difficult. As suggested above, one does not need to think of the simpler model as having the outputs coming off the states, rather one can view the outputs as still coming off the arcs, but that the output distributions happen to be the same for all arcs that start at a certain node (or that end at a certain node, if one prefers).

This suggests a general strategy. A problem with HMM models is the large number of parameters that need to be estimated to define the model, and it may not be possible to estimate them all accurately if not much data is available. A straightforward strategy for dealing with this situation is to introduce assumptions that probability distributions on certain arcs or at certain states are the same as each other. This is referred to as *parameter tying*, and one thus gets *tied states* or *tied arcs*. Another possibility for reducing the number of parameters of the model is to decide that certain things are impossible (i.e., they have probability zero), and thus to introduce structural zeroes into the model. Making some things impossible adds a lot of structure to the model, and so can greatly improve the performance of the parameter reestimation algorithm, but is only appropriate in some circumstances.

PARAMETER TYING
TIED STATES
TIED ARCS

9.4.3 Multiple input observations

We have presented the algorithms for a single input sequence. How does one train over multiple inputs? For the kind of HMM we have been assuming, where every state is connected to every other state (with a non-zero transition probability – what is sometimes called an *ergodic model* – then there is a simple solution: we simply concatenate all the observation sequences and train on them as one long input. The only real disadvantage to this is that we do not get sufficient data to be able to reestimate the initial probabilities π_i successfully. However, often people use HMM models that are not fully connected. For example, people sometimes use a *feed forward model* where there is an ordered set of states and one can only proceed at each time instant to the same or a higher numbered state. If the HMM is not fully connected – it contains structural zeroes – or if we do want to be able to reestimate the initial probabilities, then we need to extend the reeestimation formulae to work with a sequence of inputs. Provided that we assume that the inputs are independent, this is straightforward. We will

ERGODIC MODEL
FEED FORWARD MODEL

not present the formulas here, but we do present the analogous formulas for the PCFG case in Section 11.3.4.

9.4.4 Initialization of parameter values

The reestimation process only guarantees that we will find a local maximum. If we would rather find the global maximum, one approach is to try to start the HMM in a region of the parameter space that is near the global maximum. One can do this by trying to roughly estimate good values for the parameters, rather than setting them randomly. In practice, good initial estimates for the output parameters $B = \{b_{ijk}\}$ turn out to be particularly important, while random initial estimates for the parameters A and Π are normally satisfactory.

9.5 Further Reading

The Viterbi algorithm was first described in (Viterbi 1967). The mathematical theory behind Hidden Markov Models was developed by Baum and his colleagues in the late sixties and early seventies (Baum et al. 1970), and advocated for use in speech recognition in lectures by Jack Ferguson from the Institute for Defense Analyses. It was applied to speech processing in the 1970s by Baker at CMU (Baker 1975), and by Jelinek and colleagues at IBM (Jelinek et al. 1975; Jelinek 1976), and then later found its way at IBM and elsewhere into use for other kinds of language modeling, such as part of speech tagging.

There are many good references on HMM algorithms (within the context of speech recognition), including (Levinson et al. 1983; Knill and Young 1997; Jelinek 1997). Particularly well-known are (Rabiner 1989; Rabiner and Juang 1993). They consider continuous HMMs (where the output is real valued) as well as the discrete HMMs we have considered here, contain information on applications of HMMs to speech recognition and may also be consulted for fairly comprehensive references on the development and the use of HMMs. Our presentation of HMMs is however most closely based on that of (Paul 1990).

Within the chapter, we have assumed a fixed HMM architecture, and have just gone about learning optimal parameters for the HMM within that architecture. However, what size and shape of HMM should one choose for a new problem? Sometimes the nature of the problem determines the architecture, as in the applications of HMMs to tagging that we discuss in the next chapter. For circumstances when this is not the case, there has been some work on learning an appropriate HMM structure on the principle of trying to find the most compact HMM that can adequately describe the data (Stolcke and Omohundro 1993).

HMMs are widely used to analyze gene sequences in bioinformatics. See for instance (Baldi and Brunak 1998; Durbin et al. 1998). As linguists, we find it a little hard to take seriously problems over an alphabet of four symbols, but bioinformatics is a well-funded domain to which you can apply your new skills in Hidden Markov Modeling!