CHAPTER

9 Hidden Markov Models

Her sister was called Tatiana.
For the first time with such a name
the tender pages of a novel,
we'll whimsically grace.
Pushkin, *Eugene Onegin*, in the Nabokov translation

Alexander Pushkin's novel in verse, *Eugene Onegin*, serialized in the early 19th century, tells of the young dandy Onegin, his rejection of the love of young Tatiana, his duel with his friend Lenski, and his later regret for both mistakes. But the novel is mainly beloved for its style and structure rather than its plot. Among other interesting structural innovations, the novel is written in a form now known as the *Onegin stanza*, iambic tetrameter with an unusual rhyme scheme. These elements have caused complications and controversy in its translation into other languages. Many of the translations have been in verse, but Nabokov famously translated it strictly literally into English prose. The issue of its translation and the tension between literal and verse translations have inspired much commentary—see, for example, Hofstadter (1997).

In 1913, A. A. Markov asked a less controversial question about Pushkin's text: could we use frequency counts from the text to help compute the probability that the next letter in sequence would be a vowel? In this chapter we introduce a descendant of Markov's model that is a key model for language processing, the **hidden Markov model** or **HMM**.

sequence model

The HMM is a **sequence model**. A sequence model or **sequence classifier** is a model whose job is to assign a label or class to each unit in a sequence, thus mapping a sequence of observations to a sequence of labels. An HMM is a probabilistic sequence model: given a sequence of units (words, letters, morphemes, sentences, whatever), they compute a probability distribution over possible sequences of labels and choose the best label sequence.

Sequence labeling tasks come up throughout speech and language processing, a fact that isn't too surprising if we consider that language consists of sequences at many representational levels. These include part-of-speech tagging (Chapter 10) named entity tagging (Chapter 20), and speech recognition (Chapter 31) among others.

In this chapter we present the mathematics of the HMM, beginning with the Markov chain and then including the main three constituent algorithms: the **Viterbi** algorithm, the **Forward** algorithm, and the **Baum-Welch** or EM algorithm for unsupervised (or semi-supervised) learning. In the following chapter we'll see the HMM applied to the task of part-of-speech tagging.

9.1 Markov Chains

The hidden Markov model is one of the most important machine learning models in speech and language processing. To define it properly, we need to first introduce the **Markov chain**, sometimes called the **observed Markov model**. Markov chains and hidden Markov models are both extensions of the finite automata of Chapter 3. Recall that a **weighted finite automaton** is defined by a set of states and a set of transitions between states, with each arc associated with a weight. A **Markov chain** is a special case of a weighted automaton in which weights are probabilities (the probabilities on all arcs leaving a node must sum to 1) and in which the input sequence uniquely determines which states the automaton will go through. Because it can't represent inherently ambiguous problems, a Markov chain is only useful for assigning probabilities to unambiguous sequences.

Markov chain

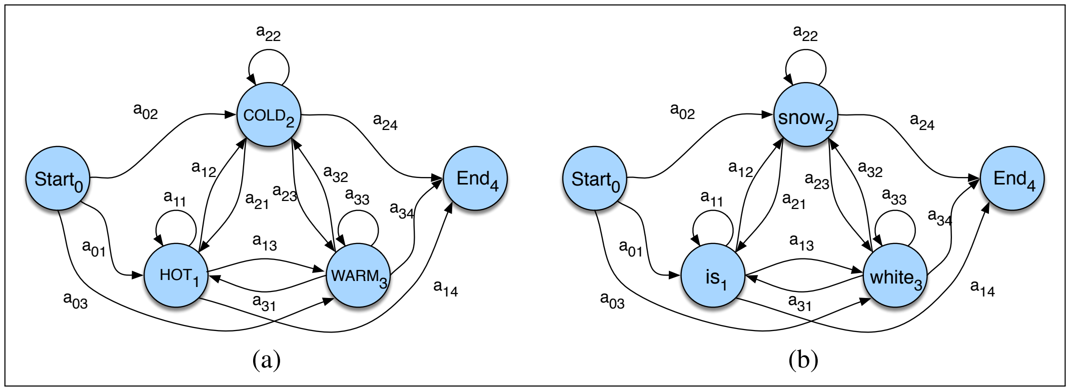

Figure 9.1 A Markov chain for weather (a) and one for words (b). A Markov chain is specified by the structure, the transition between states, and the start and end states.

Figure 9.1a shows a Markov chain for assigning a probability to a sequence of weather events, for which the vocabulary consists of HOT, COLD, and WARM. Figure 9.1b shows another simple example of a Markov chain for assigning a probability to a sequence of words $w_1...w_n$. This Markov chain should be familiar; in fact, it represents a bigram language model. Given the two models in Fig. 9.1, we can assign a probability to any sequence from our vocabulary. We go over how to do this shortly.

First, let's be more formal and view a Markov chain as a kind of probabilistic **graphical model**: a way of representing probabilistic assumptions in a graph. A Markov chain is specified by the following components:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{01} a_{02} \ldots a_{n1} \ldots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
q_0, q_F	a special **start state** and **end (final) state** that are not associated with observations

Figure 9.1 shows that we represent the states (including start and end states) as nodes in the graph, and the transitions as edges between nodes.

A Markov chain embodies an important assumption about these probabilities. In a **first-order** Markov chain, the probability of a particular state depends only on the

First-order Markov chain

previous state:

$$\textbf{Markov Assumption:} \quad P(q_i|q_1...q_{i-1}) = P(q_i|q_{i-1}) \tag{9.1}$$

Note that because each a_{ij} expresses the probability $p(q_j|q_i)$, the laws of probability require that the values of the outgoing arcs from a given state must sum to 1:

$$\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i \tag{9.2}$$

An alternative representation that is sometimes used for Markov chains doesn't rely on a start or end state, instead representing the distribution over initial states and accepting states explicitly:

$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$
$QA = \{q_x, q_y...\}$	a set $QA \subset Q$ of legal **accepting states**

Thus, the probability of state 1 being the first state can be represented either as a_{01} or as π_1. Note that because each π_i expresses the probability $p(q_i|START)$, all the π probabilities must sum to 1:

$$\sum_{i=1}^{n} \pi_i = 1 \tag{9.3}$$

Before you go on, use the sample probabilities in Fig. 9.2b to compute the probability of each of the following sequences:

(9.4) hot hot hot hot

(9.5) cold hot cold hot

What does the difference in these probabilities tell you about a real-world weather fact encoded in Fig. 9.2b?

9.2 The Hidden Markov Model

A Markov chain is useful when we need to compute a probability for a sequence of events that we can observe in the world. In many cases, however, the events we are interested in may not be directly observable in the world. For example, in Chapter 10we'll introduce the task of part-of-speech tagging, assigning tags like Noun and Verb to words.

we didn't observe part-of-speech tags in the world; we saw words and had to infer the correct tags from the word sequence. We call the part-of-speech tags **hidden** because they are not observed. The same architecture comes up in speech recognition; in that case we see acoustic events in the world and have to infer the presence of "hidden" words that are the underlying causal source of the acoustics. A **hidden Markov model** (**HMM**) allows us to talk about both *observed* events (like words

Hidden Markov model

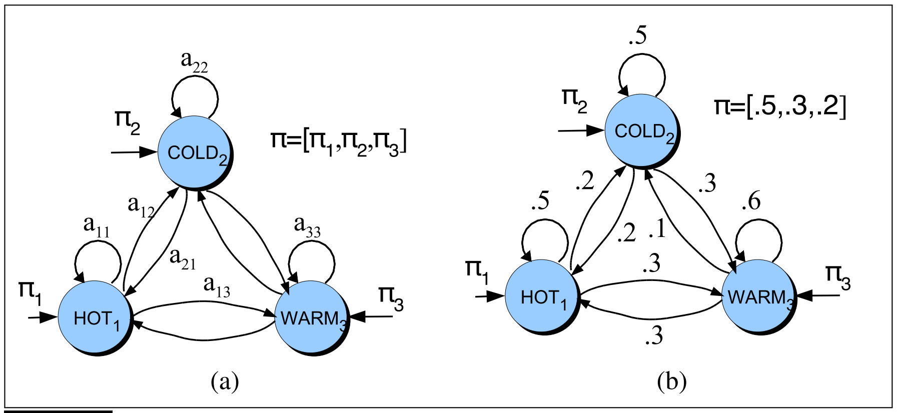

Figure 9.2 Another representation of the same Markov chain for weather shown in Fig. 9.1. Instead of using a special start state with a_{01} transition probabilities, we use the π vector, which represents the distribution over starting state probabilities. The figure in (b) shows sample probabilities.

that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in our probabilistic model.

To exemplify these models, we'll use a task conceived of by Jason Eisner (2002). Imagine that you are a climatologist in the year 2799 studying the history of global warming. You cannot find any records of the weather in Baltimore, Maryland, for the summer of 2007, but you do find Jason Eisner's diary, which lists how many ice creams Jason ate every day that summer. Our goal is to use these observations to estimate the temperature every day. We'll simplify this weather task by assuming there are only two kinds of days: cold (C) and hot (H). So the Eisner task is as follows:

> Given a sequence of observations O, each observation an integer corresponding to the number of ice creams eaten on a given day, figure out the correct 'hidden' sequence Q of weather states (H or C) which caused Jason to eat the ice cream.

Let's begin with a formal definition of a hidden Markov model, focusing on how it differs from a Markov chain. An HMM is specified by the following components:

$Q = q_1 q_2 \dots q_N$	a set of N **states**
$A = a_{11} a_{12} \dots a_{n1} \dots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \dots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state i
q_0, q_F	a special **start state** and **end (final) state** that are not associated with observations, together with transition probabilities $a_{01} a_{02} \dots a_{0n}$ out of the start state and $a_{1F} a_{2F} \dots a_{nF}$ into the end state

As we noted for Markov chains, an alternative representation that is sometimes used for HMMs doesn't rely on a start or end state, instead representing the distribution over initial and accepting states explicitly. We don't use the π notation in this textbook, but you may see it in the literature[1]:

$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$
$QA = \{q_x, q_y ...\}$	a set $QA \subset Q$ of legal **accepting states**

A first-order hidden Markov model instantiates two simplifying assumptions. First, as with a first-order Markov chain, the probability of a particular state depends only on the previous state:

$$\textbf{Markov Assumption:}\quad P(q_i|q_1...q_{i-1}) = P(q_i|q_{i-1}) \tag{9.6}$$

Second, the probability of an output observation o_i depends only on the state that produced the observation q_i and not on any other states or any other observations:

$$\textbf{Output Independence:}\quad P(o_i|q_1 \ldots q_i, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i|q_i) \tag{9.7}$$

Figure 9.3 shows a sample HMM for the ice cream task. The two hidden states (H and C) correspond to hot and cold weather, and the observations (drawn from the alphabet $O = \{1,2,3\}$) correspond to the number of ice creams eaten by Jason on a given day.

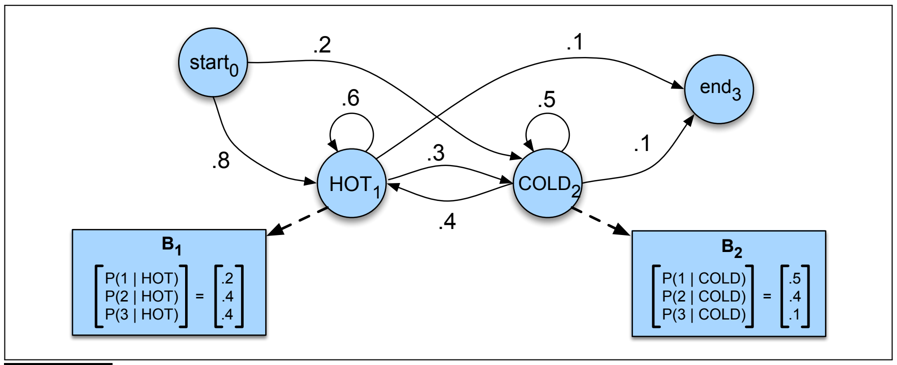

Figure 9.3 A hidden Markov model for relating numbers of ice creams eaten by Jason (the observations) to the weather (H or C, the hidden variables).

Notice that in the HMM in Fig. 9.3, there is a (non-zero) probability of transitioning between any two states. Such an HMM is called a **fully connected** or **ergodic HMM**. Sometimes, however, we have HMMs in which many of the transitions between states have zero probability. For example, in **left-to-right** (also called **Bakis**) HMMs, the state transitions proceed from left to right, as shown in Fig. 9.4. In a Bakis HMM, no transitions go from a higher-numbered state to a lower-numbered state (or, more accurately, any transitions from a higher-numbered state to a lower-numbered state have zero probability). Bakis HMMs are generally used to model temporal processes like speech; we show more of them in Chapter 31.

Ergodic HMM

Bakis network

[1] It is also possible to have HMMs without final states or explicit accepting states. Such HMMs define a set of probability distributions, one distribution per observation sequence length, just as language models do when they don't have explicit end symbols. This isn't a problem since for most tasks in speech and language processing the lengths of the observations are fixed.

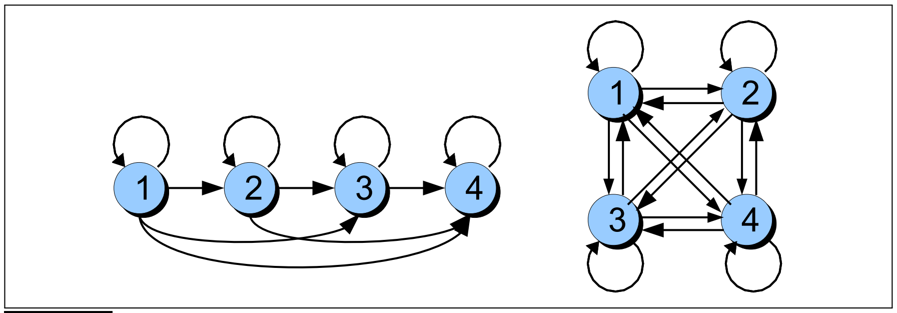

Figure 9.4 Two 4-state hidden Markov models; a left-to-right (Bakis) HMM on the left and a fully connected (ergodic) HMM on the right. In the Bakis model, all transitions not shown have zero probability.

Now that we have seen the structure of an HMM, we turn to algorithms for computing things with them. An influential tutorial by Rabiner (1989), based on tutorials by Jack Ferguson in the 1960s, introduced the idea that hidden Markov models should be characterized by **three fundamental problems**:

Problem 1 (Likelihood):	Given an HMM $\lambda = (A,B)$ and an observation sequence O, determine the likelihood $P(O\|\lambda)$.
Problem 2 (Decoding):	Given an observation sequence O and an HMM $\lambda = (A,B)$, discover the best hidden state sequence Q.
Problem 3 (Learning):	Given an observation sequence O and the set of states in the HMM, learn the HMM parameters A and B.

We already saw an example of Problem 2 in Chapter 10. In the next three sections we introduce all three problems more formally.

9.3 Likelihood Computation: The Forward Algorithm

Our first problem is to compute the likelihood of a particular observation sequence. For example, given the ice-cream eating HMM in Fig. 9.3, what is the probability of the sequence *3 1 3*? More formally:

> **Computing Likelihood:** Given an HMM $\lambda = (A,B)$ and an observation sequence O, determine the likelihood $P(O|\lambda)$.

For a Markov chain, where the surface observations are the same as the hidden events, we could compute the probability of *3 1 3* just by following the states labeled *3 1 3* and multiplying the probabilities along the arcs. For a hidden Markov model, things are not so simple. We want to determine the probability of an ice-cream observation sequence like *3 1 3*, but we don't know what the hidden state sequence is!

Let's start with a slightly simpler situation. Suppose we already knew the weather and wanted to predict how much ice cream Jason would eat. This is a useful part of many HMM tasks. For a given hidden state sequence (e.g., *hot hot cold*), we can easily compute the output likelihood of *3 1 3*.

Let's see how. First, recall that for hidden Markov models, each hidden state produces only a single observation. Thus, the sequence of hidden states and the sequence of observations have the same length.[2]

Given this one-to-one mapping and the Markov assumptions expressed in Eq. 9.6, for a particular hidden state sequence $Q = q_0, q_1, q_2, ..., q_T$ and an observation sequence $O = o_1, o_2, ..., o_T$, the likelihood of the observation sequence is

$$P(O|Q) = \prod_{i=1}^{T} P(o_i|q_i) \tag{9.8}$$

The computation of the forward probability for our ice-cream observation *3 1 3* from one possible hidden state sequence *hot hot cold* is shown in Eq. 9.9. Figure 9.5 shows a graphic representation of this computation.

$$P(3\ 1\ 3|\text{hot hot cold}) \quad = \quad P(3|\text{hot}) \times P(1|\text{hot}) \times P(3|\text{cold}) \tag{9.9}$$

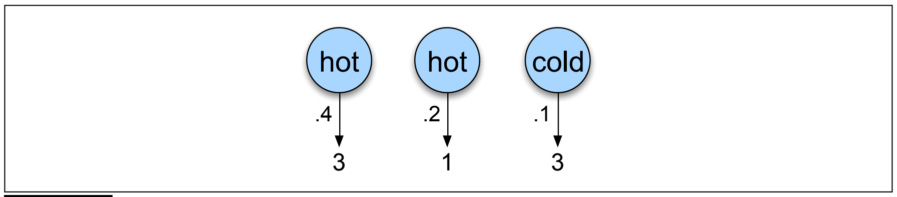

Figure 9.5 The computation of the observation likelihood for the ice-cream events *3 1 3* given the hidden state sequence *hot hot cold*.

But of course, we don't actually know what the hidden state (weather) sequence was. We'll need to compute the probability of ice-cream events *3 1 3* instead by summing over all possible weather sequences, weighted by their probability. First, let's compute the joint probability of being in a particular weather sequence Q and generating a particular sequence O of ice-cream events. In general, this is

$$P(O,Q) = P(O|Q) \times P(Q) = \prod_{i=1}^{T} P(o_i|q_i) \times \prod_{i=1}^{T} P(q_i|q_{i-1}) \tag{9.10}$$

The computation of the joint probability of our ice-cream observation *3 1 3* and one possible hidden state sequence *hot hot cold* is shown in Eq. 9.11. Figure 9.6 shows a graphic representation of this computation.

$$\begin{aligned} P(3\ 1\ 3, \text{hot hot cold}) \quad &= \quad P(\text{hot}|\text{start}) \times P(\text{hot}|\text{hot}) \times P(\text{cold}|\text{hot}) \\ & \quad\quad \times P(3|\text{hot}) \times P(1|\text{hot}) \times P(3|\text{cold}) \end{aligned} \tag{9.11}$$

Now that we know how to compute the joint probability of the observations with a particular hidden state sequence, we can compute the total probability of the observations just by summing over all possible hidden state sequences:

$$P(O) = \sum_{Q} P(O,Q) = \sum_{Q} P(O|Q)P(Q) \tag{9.12}$$

[2] In a variant of HMMs called **segmental HMMs** (in speech recognition) or **semi-HMMs** (in text processing) this one-to-one mapping between the length of the hidden state sequence and the length of the observation sequence does not hold.

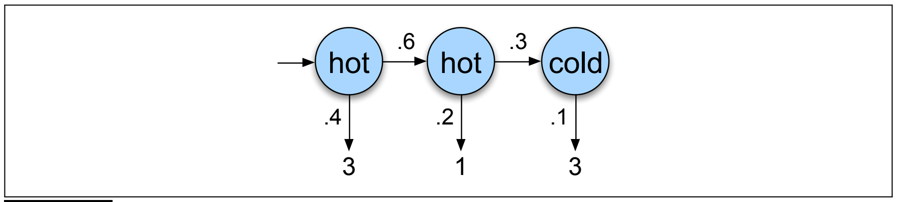

Figure 9.6 The computation of the joint probability of the ice-cream events *3 1 3* and the hidden state sequence *hot hot cold*.

For our particular case, we would sum over the eight 3-event sequences *cold cold cold*, *cold cold hot*, that is,

$$P(3\ 1\ 3) = P(3\ 1\ 3, \text{cold cold cold}) + P(3\ 1\ 3, \text{cold cold hot}) + P(3\ 1\ 3, \text{hot hot cold}) + ...$$

For an HMM with N hidden states and an observation sequence of T observations, there are N^T possible hidden sequences. For real tasks, where N and T are both large, N^T is a very large number, so we cannot compute the total observation likelihood by computing a separate observation likelihood for each hidden state sequence and then summing them.

Forward algorithm

Instead of using such an extremely exponential algorithm, we use an efficient $O(N^2T)$ algorithm called the **forward algorithm**. The forward algorithm is a kind of **dynamic programming** algorithm, that is, an algorithm that uses a table to store intermediate values as it builds up the probability of the observation sequence. The forward algorithm computes the observation probability by summing over the probabilities of all possible hidden state paths that could generate the observation sequence, but it does so efficiently by implicitly folding each of these paths into a single **forward trellis**.

Figure 9.7 shows an example of the forward trellis for computing the likelihood of *3 1 3* given the hidden state sequence *hot hot cold*.

Each cell of the forward algorithm trellis $\alpha_t(j)$ represents the probability of being in state j after seeing the first t observations, given the automaton λ. The value of each cell $\alpha_t(j)$ is computed by summing over the probabilities of every path that could lead us to this cell. Formally, each cell expresses the following probability:

$$\alpha_t(j) = P(o_1, o_2 \dots o_t, q_t = j|\lambda) \tag{9.13}$$

Here, $q_t = j$ means "the tth state in the sequence of states is state j". We compute this probability $\alpha_t(j)$ by summing over the extensions of all the paths that lead to the current cell. For a given state q_j at time t, the value $\alpha_t(j)$ is computed as

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \tag{9.14}$$

The three factors that are multiplied in Eq. 9.14 in extending the previous paths to compute the forward probability at time t are

$\alpha_{t-1}(i)$	the **previous forward path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

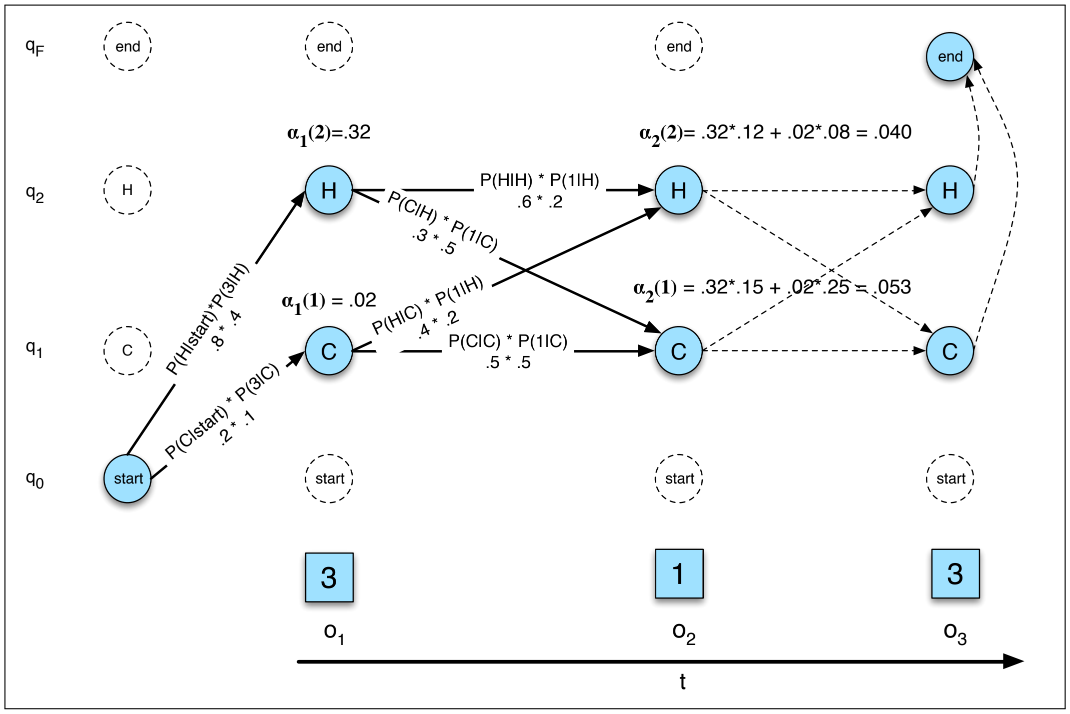

Figure 9.7 The forward trellis for computing the total observation likelihood for the ice-cream events *3 1 3*. Hidden states are in circles, observations in squares. White (unfilled) circles indicate illegal transitions. The figure shows the computation of $\alpha_t(j)$ for two states at two time steps. The computation in each cell follows Eq. 9.14: $\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i)a_{ij}b_j(o_t)$. The resulting probability expressed in each cell is Eq. 9.13: $\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)$.

Consider the computation in Fig. 9.7 of $\alpha_2(2)$, the forward probability of being at time step 2 in state 2 having generated the partial observation *3 1*. We compute by extending the α probabilities from time step 1, via two paths, each extension consisting of the three factors above: $\alpha_1(1) \times P(H|H) \times P(1|H)$ and $\alpha_1(2) \times P(H|C) \times P(1|H)$.

Figure 9.8 shows another visualization of this induction step for computing the value in one new cell of the trellis.

We give two formal definitions of the forward algorithm: the pseudocode in Fig. 9.9 and a statement of the definitional recursion here.

1. Initialization:

$$\alpha_1(j) \;=\; a_{0j}b_j(o_1) \quad 1 \le j \le N \tag{9.15}$$

2. Recursion (since states 0 and F are non-emitting):

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i)a_{ij}b_j(o_t); \quad 1 \le j \le N, 1 < t \le T \tag{9.16}$$

3. Termination:

$$P(O|\lambda) = \alpha_T(q_F) = \sum_{i=1}^{N} \alpha_T(i)\, a_{iF} \tag{9.17}$$

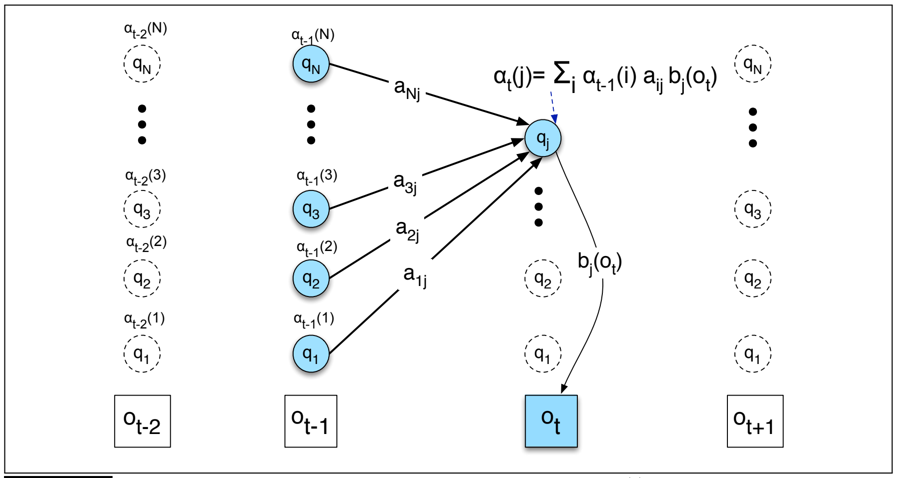

Figure 9.8 Visualizing the computation of a single element $\alpha_t(i)$ in the trellis by summing all the previous values α_{t-1}, weighted by their transition probabilities a, and multiplying by the observation probability $b_i(o_{t+1})$. For many applications of HMMs, many of the transition probabilities are 0, so not all previous states will contribute to the forward probability of the current state. Hidden states are in circles, observations in squares. Shaded nodes are included in the probability computation for $\alpha_t(i)$. Start and end states are not shown.

```
function FORWARD(observations of len T, state-graph of len N) returns forward-prob

  create a probability matrix forward[N+2,T]
  for each state s from 1 to N do                          ; initialization step
      $forward[s,1] \leftarrow a_{0,s} * b_s(o_1)$
  for each time step t from 2 to T do                       ; recursion step
    for each state s from 1 to N do
          $forward[s,t] \leftarrow \sum_{s'=1}^{N} forward[s',t-1] * a_{s',s} * b_s(o_t)$
  $forward[q_F,T] \leftarrow \sum_{s=1}^{N} forward[s,T] * a_{s,q_F}$          ; termination step
 return $forward[q_F,T]$
```

Figure 9.9 The forward algorithm. We've used the notation $forward[s,t]$ to represent $\alpha_t(s)$.

9.4 Decoding: The Viterbi Algorithm

Decoding
Decoder

For any model, such as an HMM, that contains hidden variables, the task of determining which sequence of variables is the underlying source of some sequence of observations is called the **decoding** task. In the ice-cream domain, given a sequence of ice-cream observations *3 1 3* and an HMM, the task of the **decoder** is to find the best hidden weather sequence (*H H H*). More formally,

Decoding: Given as input an HMM $\lambda = (A,B)$ and a sequence of observations $O = o_1, o_2, ..., o_T$, find the most probable sequence of states $Q = q_1 q_2 q_3 \dots q_T$.

We might propose to find the best sequence as follows: For each possible hidden state sequence (*HHH*, *HHC*, *HCH*, etc.), we could run the forward algorithm and compute the likelihood of the observation sequence given that hidden state sequence. Then we could choose the hidden state sequence with the maximum observation likelihood. It should be clear from the previous section that we cannot do this because there are an exponentially large number of state sequences.

Viterbi algorithm

Instead, the most common decoding algorithms for HMMs is the **Viterbi algorithm**. Like the forward algorithm, **Viterbi** is a kind of **dynamic programming** that makes uses of a dynamic programming trellis. Viterbi also strongly resembles another dynamic programming variant, the **minimum edit distance** algorithm of Chapter 3.

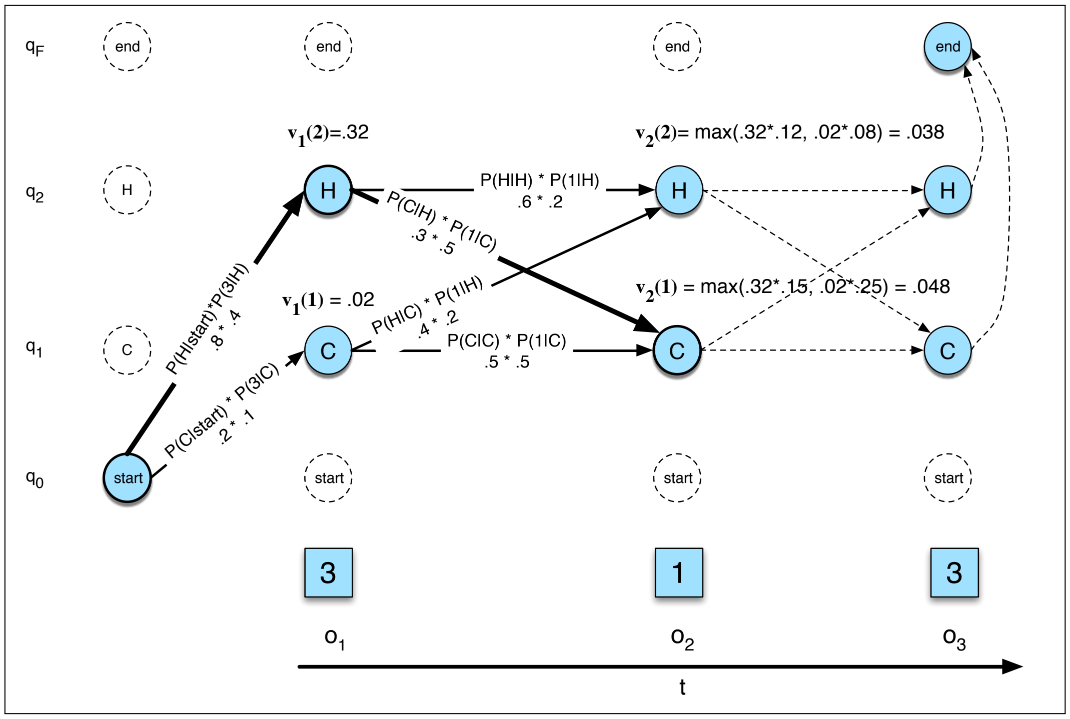

Figure 9.10 The Viterbi trellis for computing the best path through the hidden state space for the ice-cream eating events *3 1 3*. Hidden states are in circles, observations in squares. White (unfilled) circles indicate illegal transitions. The figure shows the computation of $v_t(j)$ for two states at two time steps. The computation in each cell follows Eq. 9.19: $v_t(j) = \max_{1\leq i\leq N-1} v_{t-1}(i)\, a_{ij}\, b_j(o_t)$. The resulting probability expressed in each cell is Eq. 9.18: $v_t(j) = P(q_0, q_1, \ldots, q_{t-1}, o_1, o_2, \ldots, o_t, q_t = j|\lambda)$.

Figure 9.10 shows an example of the Viterbi trellis for computing the best hidden state sequence for the observation sequence *3 1 3*. The idea is to process the observation sequence left to right, filling out the trellis. Each cell of the trellis, $v_t(j)$, represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_0, q_1, ..., q_{t-1}$, given the automaton λ. The value of each cell $v_t(j)$ is computed by recursively taking the most probable path that could lead us to this cell. Formally, each cell expresses the probability

$$v_t(j) = \max_{q_0, q_1, \ldots, q_{t-1}} P(q_0, q_1 ... q_{t-1}, o_1, o_2 \ldots o_t, q_t = j|\lambda) \tag{9.18}$$

Note that we represent the most probable path by taking the maximum over all possible previous state sequences $\max_{q_0,q_1,\ldots,q_{t-1}}$. Like other dynamic programming algorithms, Viterbi fills each cell recursively. Given that we had already computed the probability of being in every state at time $t-1$, we compute the Viterbi probability by taking the most probable of the extensions of the paths that lead to the current cell. For a given state q_j at time t, the value $v_t(j)$ is computed as

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t) \tag{9.19}$$

The three factors that are multiplied in Eq. 9.19 for extending the previous paths to compute the Viterbi probability at time t are

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

```
function VITERBI(observations of len T, state-graph of len N) returns best-path

  create a path probability matrix viterbi[N+2,T]
  for each state s from 1 to N do                          ; initialization step
      $viterbi[s,1] \leftarrow a_{0,s} * b_s(o_1)$
      $backpointer[s,1] \leftarrow 0$
  for each time step t from 2 to T do                      ; recursion step
     for each state s from 1 to N do
      $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
      $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s}$
 $viterbi[q_F,T] \leftarrow \max_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$          ; termination step
 $backpointer[q_F,T] \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$     ; termination step
  return the backtrace path by following backpointers to states back in
          time from $backpointer[q_F,T]$
```

Figure 9.11 Viterbi algorithm for finding optimal sequence of hidden states. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence. Note that states 0 and q_F are non-emitting.

Figure 9.11 shows pseudocode for the Viterbi algorithm. Note that the Viterbi algorithm is identical to the forward algorithm except that it takes the **max** over the previous path probabilities whereas the forward algorithm takes the **sum**. Note also that the Viterbi algorithm has one component that the forward algorithm doesn't have: **backpointers**. The reason is that while the forward algorithm needs to produce an observation likelihood, the Viterbi algorithm must produce a probability and also the most likely state sequence. We compute this best state sequence by keeping track of the path of hidden states that led to each state, as suggested in Fig. 9.12, and then at the end backtracing the best path to the beginning (the Viterbi **backtrace**).

Viterbi backtrace

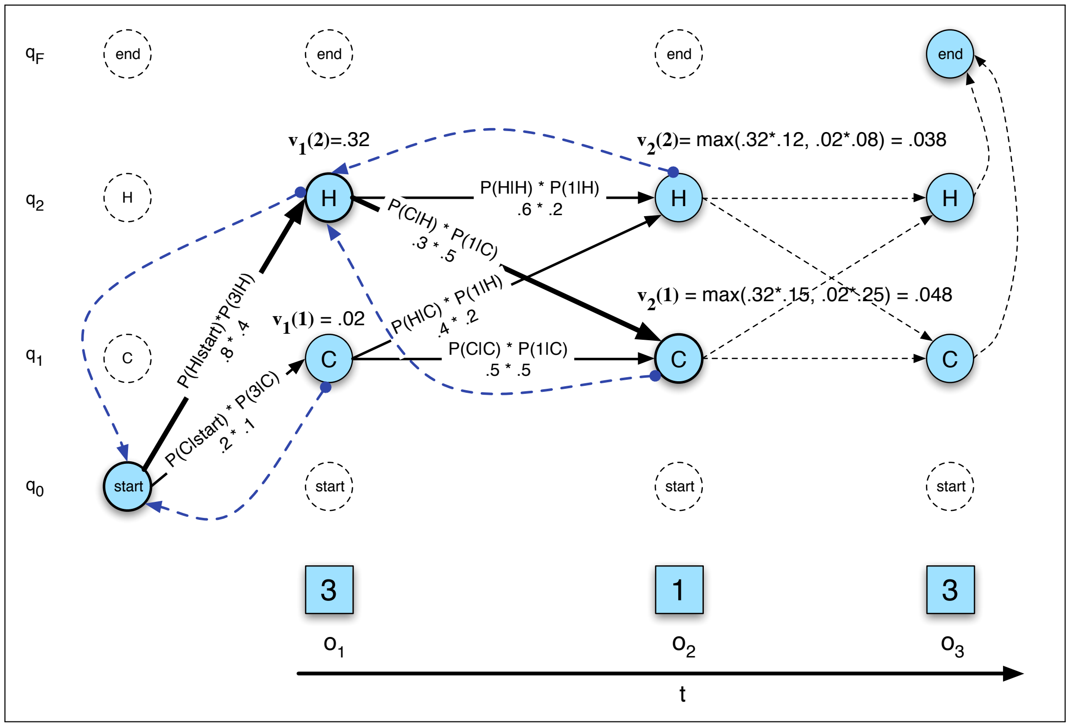

Figure 9.12 The Viterbi backtrace. As we extend each path to a new state account for the next observation, we keep a backpointer (shown with broken lines) to the best path that led us to this state.

Finally, we can give a formal definition of the Viterbi recursion as follows:

1. **Initialization:**

$$v_1(j) = a_{0j}b_j(o_1) \quad 1 \le j \le N \tag{9.20}$$
$$bt_1(j) = 0 \tag{9.21}$$

2. **Recursion** (recall that states 0 and q_F are non-emitting):

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\,a_{ij}\,b_j(o_t); \quad 1 \le j \le N, 1 < t \le T \tag{9.22}$$
$$bt_t(j) = \operatorname{argmax}_{i=1}^{N} v_{t-1}(i)\,a_{ij}\,b_j(o_t); \quad 1 \le j \le N, 1 < t \le T \tag{9.23}$$

3. **Termination:**

$$\text{The best score:} \quad P* = v_T(q_F) = \max_{i=1}^{N} v_T(i) * a_{iF} \tag{9.24}$$
$$\text{The start of backtrace:} \quad q_T* = bt_T(q_F) = \operatorname{argmax}_{i=1}^{N} v_T(i) * a_{iF} \tag{9.25}$$

9.5 HMM Training: The Forward-Backward Algorithm

We turn to the third problem for HMMs: learning the parameters of an HMM, that is, the *A* and *B* matrices. Formally,

Learning: Given an observation sequence O and the set of possible states in the HMM, learn the HMM parameters A and B.

The input to such a learning algorithm would be an unlabeled sequence of observations O and a vocabulary of potential hidden states Q. Thus, for the ice cream task, we would start with a sequence of observations $O = \{1,3,2,...,\}$ and the set of hidden states H and C. For the part-of-speech tagging task we introduce in the next chapter, we would start with a sequence of word observations $O = \{w_1, w_2, w_3 \dots\}$ and a set of hidden states corresponding to parts of speech *Noun, Verb, Adjective,...* and so on.

Forward-backward
Baum-Welch
EM

The standard algorithm for HMM training is the **forward-backward**, or **Baum-Welch** algorithm (Baum, 1972), a special case of the **Expectation-Maximization** or **EM** algorithm (Dempster et al., 1977). The algorithm will let us train both the transition probabilities A and the emission probabilities B of the HMM. Crucially, EM is an *iterative* algorithm. It works by computing an initial estimate for the probabilities, then using those estimates to computing a better estimate, and so on, iteratively improving the probabilities that it learns.

Let us begin by considering the much simpler case of training a Markov chain rather than a hidden Markov model. Since the states in a Markov chain are observed, we can run the model on the observation sequence and directly see which path we took through the model and which state generated each observation symbol. A Markov chain of course has no emission probabilities B (alternatively, we could view a Markov chain as a degenerate hidden Markov model where all the b probabilities are 1.0 for the observed symbol and 0 for all other symbols). Thus, the only probabilities we need to train are the transition probability matrix A.

We get the maximum likelihood estimate of the probability a_{ij} of a particular transition between states i and j by counting the number of times the transition was taken, which we could call $C(i \to j)$, and then normalizing by the total count of all times we took any transition from state i:

$$a_{ij} = \frac{C(i \to j)}{\sum_{q \in Q} C(i \to q)} \tag{9.26}$$

We can directly compute this probability in a Markov chain because we know which states we were in. For an HMM, we cannot compute these counts directly from an observation sequence since we don't know which path of states was taken through the machine for a given input. The Baum-Welch algorithm uses two neat intuitions to solve this problem. The first idea is to *iteratively* estimate the counts. We will start with an estimate for the transition and observation probabilities and then use these estimated probabilities to derive better and better probabilities. The second idea is that we get our estimated probabilities by computing the forward probability for an observation and then dividing that probability mass among all the different paths that contributed to this forward probability.

Backward probability

To understand the algorithm, we need to define a useful probability related to the forward probability and called the **backward probability**.

The backward probability β is the probability of seeing the observations from time $t+1$ to the end, given that we are in state i at time t (and given the automaton λ):

$$\beta_t(i) = P(o_{t+1}, o_{t+2} \dots o_T | q_t = i, \lambda) \tag{9.27}$$

It is computed inductively in a similar manner to the forward algorithm.

1. **Initialization:**

$$\beta_T(i) \;=\; a_{iF}, \quad 1 \le i \le N \tag{9.28}$$

2. **Recursion** (again since states 0 and q_F are non-emitting):

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j), \quad 1 \le i \le N, 1 \le t < T \tag{9.29}$$

3. **Termination:**

$$P(O|\lambda) = \alpha_T(q_F) = \beta_1(q_0) = \sum_{j=1}^{N} a_{0j}\, b_j(o_1)\, \beta_1(j) \tag{9.30}$$

Figure 9.13 illustrates the backward induction step.

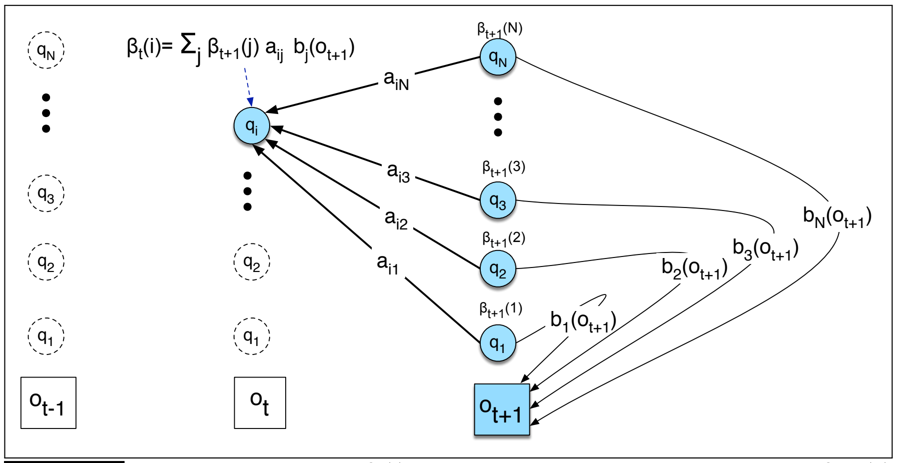

Figure 9.13 The computation of $\beta_t(i)$ by summing all the successive values $\beta_{t+1}(j)$ weighted by their transition probabilities a_{ij} and their observation probabilities $b_j(o_{t+1})$. Start and end states not shown.

We are now ready to understand how the forward and backward probabilities can help us compute the transition probability a_{ij} and observation probability $b_i(o_t)$ from an observation sequence, even though the actual path taken through the machine is hidden.

Let's begin by seeing how to estimate $\hat{a}_{ij}$ by a variant of Eq. 9.26:

$$\hat{a}_{ij} = \frac{\text{expected number of transitions from state } i \text{ to state } j}{\text{expected number of transitions from state } i} \tag{9.31}$$

How do we compute the numerator? Here's the intuition. Assume we had some estimate of the probability that a given transition $i \to j$ was taken at a particular point in time t in the observation sequence. If we knew this probability for each particular time t, we could sum over all times t to estimate the total count for the transition $i \to j$.

More formally, let's define the probability ξ_t as the probability of being in state i at time t and state j at time $t+1$, given the observation sequence and of course the model:

$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j | O, \lambda) \tag{9.32}$$

To compute ξ_t, we first compute a probability which is similar to ξ_t, but differs in including the probability of the observation; note the different conditioning of O from Eq. 9.32:

$$\text{not-quite-}\xi_t(i,j) = P(q_t = i, q_{t+1} = j, O|\lambda) \tag{9.33}$$

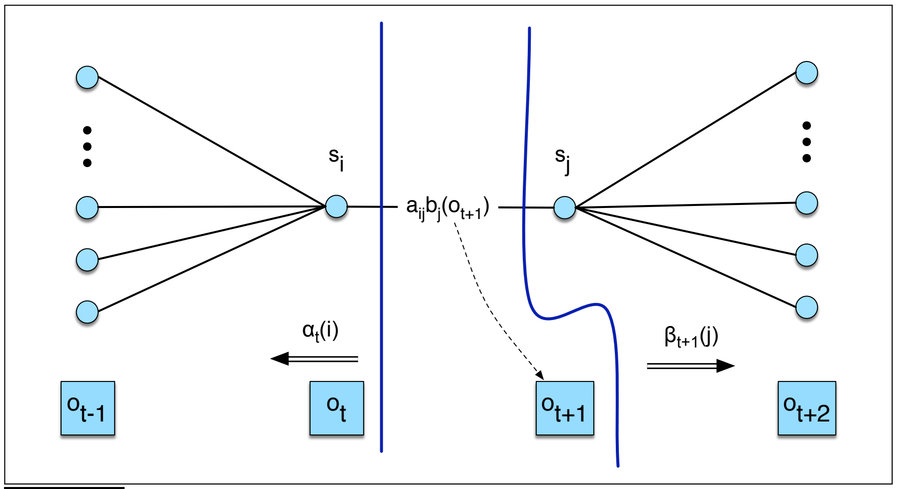

Figure 9.14 Computation of the joint probability of being in state i at time t and state j at time $t+1$. The figure shows the various probabilities that need to be combined to produce $P(q_t = i, q_{t+1} = j, O|\lambda)$: the α and β probabilities, the transition probability a_{ij} and the observation probability $b_j(o_{t+1})$. After Rabiner (1989) which is ©1989 IEEE.

Figure 9.14 shows the various probabilities that go into computing not-quite-ξ_t: the transition probability for the arc in question, the α probability before the arc, the β probability after the arc, and the observation probability for the symbol just after the arc. These four are multiplied together to produce *not-quite-*ξ_t as follows:

$$\text{not-quite-}\xi_t(i,j) = \alpha_t(i)\, a_{ij} b_j(o_{t+1})\beta_{t+1}(j) \tag{9.34}$$

To compute ξ_t from *not-quite-*ξ_t, we follow the laws of probability and divide by $P(O|\lambda)$, since

$$P(X|Y,Z) = \frac{P(X,Y|Z)}{P(Y|Z)} \tag{9.35}$$

The probability of the observation given the model is simply the forward probability of the whole utterance (or alternatively, the backward probability of the whole utterance), which can thus be computed in a number of ways:

$$P(O|\lambda) = \alpha_T(q_F) = \beta_T(q_0) = \sum_{j=1}^{N} \alpha_t(j)\beta_t(j) \tag{9.36}$$

So, the final equation for ξ_t is

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij} b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)} \tag{9.37}$$

The expected number of transitions from state i to state j is then the sum over all t of ξ. For our estimate of a_{ij} in Eq. 9.31, we just need one more thing: the total expected number of transitions from state i. We can get this by summing over all transitions out of state i. Here's the final formula for $\hat{a}_{ij}$:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)} \tag{9.38}$$

We also need a formula for recomputing the observation probability. This is the probability of a given symbol v_k from the observation vocabulary V, given a state j: $\hat{b}_j(v_k)$. We will do this by trying to compute

$$\hat{b}_j(v_k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} \tag{9.39}$$

For this, we will need to know the probability of being in state j at time t, which we will call $\gamma_t(j)$:

$$\gamma_t(j) = P(q_t = j|O,\lambda) \tag{9.40}$$

Once again, we will compute this by including the observation sequence in the probability:

$$\gamma_t(j) = \frac{P(q_t = j, O|\lambda)}{P(O|\lambda)} \tag{9.41}$$

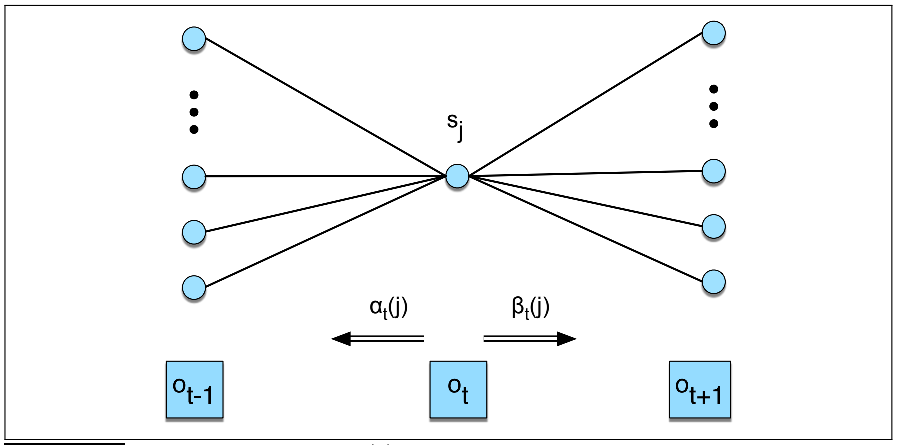

Figure 9.15 The computation of $\gamma_t(j)$, the probability of being in state j at time t. Note that γ is really a degenerate case of ξ and hence this figure is like a version of Fig. 9.14 with state i collapsed with state j. After Rabiner (1989) which is ©1989 IEEE.

As Fig. 9.15 shows, the numerator of Eq. 9.41 is just the product of the forward probability and the backward probability:

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)} \tag{9.42}$$

We are ready to compute b. For the numerator, we sum $\gamma_t(j)$ for all time steps t in which the observation o_t is the symbol v_k that we are interested in. For the denominator, we sum $\gamma_t(j)$ over all time steps t. The result is the percentage of the times that we were in state j and saw symbol v_k (the notation $\sum_{t=1 s.t. O_t=v_k}^{T}$ means "sum over all t for which the observation at time t was v_k"):

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t. O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{9.43}$$

We now have ways in Eq. 9.38 and Eq. 9.43 to *re-estimate* the transition A and observation B probabilities from an observation sequence O, assuming that we already have a previous estimate of A and B.

These re-estimations form the core of the iterative forward-backward algorithm. The forward-backward algorithm (Fig. 9.16) starts with some initial estimate of the HMM parameters $\lambda = (A, B)$. We then iteratively run two steps. Like other cases of the EM (expectation-maximization) algorithm, the forward-backward algorithm has two steps: the **expectation** step, or **E-step**, and the **maximization** step, or **M-step**.

E-step
M-step

In the E-step, we compute the expected state occupancy count γ and the expected state transition count ξ from the earlier A and B probabilities. In the M-step, we use γ and ξ to recompute new A and B probabilities.

```
function FORWARD-BACKWARD(observations of len T, output vocabulary V, hidden
state set Q) returns HMM=(A,B)

  initialize A and B
  iterate until convergence
    E-step
      $\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)}$  $\forall$ t and j
      $\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)}$  $\forall$ t, i, and j
    M-step
      $\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N}\xi_t(i,k)}$
      $\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t. O_t=v_k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)}$
  return A, B
```

Figure 9.16 The forward-backward algorithm.

Although in principle the forward-backward algorithm can do completely unsupervised learning of the A and B parameters, in practice the initial conditions are very important. For this reason the algorithm is often given extra information. For example, for speech recognition, in practice the HMM structure is often set by hand, and only the emission (B) and (non-zero) A transition probabilities are trained from a set of observation sequences O. Section **??** in Chapter 31 also discusses how initial A and B estimates are derived in speech recognition. We also show that for speech the forward-backward algorithm can be extended to inputs that are non-discrete ("continuous observation densities").

9.6 Summary

This chapter introduced the **hidden Markov model** for probabilistic **sequence classification**.

- Hidden Markov models (**HMMs**) are a way of relating a sequence of **observations** to a sequence of **hidden classes** or **hidden states** that explain the observations.
- The process of discovering the sequence of hidden states, given the sequence of observations, is known as **decoding** or **inference**. The **Viterbi** algorithm is commonly used for decoding.
- The parameters of an HMM are the *A* transition probability matrix and the *B* observation likelihood matrix. Both can be trained with the **Baum-Welch** or **forward-backward** algorithm.

Bibliographical and Historical Notes

As we discussed at the end of Chapter 4, Markov chains were first used by Markov (1913, 2006), to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a consonant.

The hidden Markov model was developed by Baum and colleagues at the Institute for Defense Analyses in Princeton (Baum and Petrie, 1966; Baum and Eagon, 1967).

The **Viterbi** algorithm was first applied to speech and language processing in the context of speech recognition by Vintsyuk (1968) but has what Kruskal (1983) calls a "remarkable history of multiple independent discovery and publication".[3] 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

The use of the term **Viterbi** is now standard for the application of dynamic programming to any kind of probabilistic maximization problem in speech and language processing. For non-probabilistic problems (such as for minimum edit distance), the plain term **dynamic programming** is often used. Forney, Jr. (1973) wrote an early survey paper that explores the origin of the Viterbi algorithm in the context of information and communications theory.

Our presentation of the idea that hidden Markov models should be characterized by three fundamental problems was modeled after an influential tutorial by Rabiner (1989), which was itself based on tutorials by Jack Ferguson of IDA in the 1960s. Jelinek (1997) and Rabiner and Juang (1993) give very complete descriptions of the

[3] Seven is pretty remarkable, but see page **??** for a discussion of the prevalence of multiple discovery.

forward-backward algorithm as applied to the speech recognition problem. Jelinek (1997) also shows the relationship between forward-backward and EM. See also the description of HMMs in other textbooks such as Manning and Schütze (1999).

Exercises

9.1 Implement the Forward algorithm and run it with the HMM in Fig. 9.3 to compute the probability of the observation sequences *331122313* and *331123312*. Which is more likely?

9.2 Implement the Viterbi algorithm and run it with the HMM in Fig. 9.3 to compute the most likely weather sequences for each of the two observation sequences above, *331122313* and *331123312*.

9.3 Extend the HMM tagger you built in Exercise 10.5 by adding the ability to make use of some unlabeled data in addition to your labeled training corpus. First acquire a large unlabeled (i.e., no part-of-speech tags) corpus. Next, implement the forward-backward training algorithm. Now start with the HMM parameters you trained on the training corpus in Exercise 10.5; call this model M_0. Run the forward-backward algorithm with these HMM parameters to label the unsupervised corpus. Now you have a new model M_1. Test the performance of M_1 on some held-out labeled data.

9.4 As a generalization of the previous homework, implement Jason Eisner's HMM tagging homework available from his webpage. His homework includes a corpus of weather and ice-cream observations, a corpus of English part-of-speech tags, and a very hand spreadsheet with exact numbers for the forward-backward algorithm that you can compare against.

CHAPTER 10 Part-of-Speech Tagging

Conjunction Junction, what's your function?
Bob Dorough, *Schoolhouse Rock, 1973*

A gnostic was seated before a grammarian. The grammarian said, 'A word must be one of three things: either it is a noun, a verb, or a particle.' The gnostic tore his robe and cried, 'Alas! Twenty years of my life and striving and seeking have gone to the winds, for I laboured greatly in the hope that there was another word outside of this. Now you have destroyed my hope.' Though the gnostic had already attained the word which was his purpose, he spoke thus in order to arouse the grammarian.
Rumi (1207–1273), *The Discourses of Rumi*, Translated by A. J. Arberry

Dionysius Thrax of Alexandria (*c.* 100 B.C.), or perhaps someone else (exact authorship being understandably difficult to be sure of with texts of this vintage), wrote a grammatical sketch of Greek (a "*technē*") that summarized the linguistic knowledge of his day. This work is the source of an astonishing proportion of modern linguistic vocabulary, including words like *syntax*, *diphthong*, *clitic*, and *analogy*. Also included are a description of eight **parts-of-speech**: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Although earlier scholars (including Aristotle as well as the Stoics) had their own lists of parts-of-speech, it was Thrax's set of eight that became the basis for practically all subsequent part-of-speech descriptions of Greek, Latin, and most European languages for the next 2000 years.

parts-of-speech

Schoolhouse Rock was a popular series of 3-minute musical animated clips first aired on television in 1973. The series was designed to inspire kids to learn multiplication tables, grammar, basic science, and history. The Grammar Rock sequence, for example, included songs about parts-of-speech, thus bringing these categories into the realm of popular culture. As it happens, Grammar Rock was remarkably traditional in its grammatical notation, including exactly eight songs about parts-of-speech. Although the list was slightly modified from Thrax's original, substituting adjective and interjection for the original participle and article, the astonishing durability of the parts-of-speech through two millenia is an indicator of both the importance and the transparency of their role in human language. Nonetheless, eight isn't very many and more recent part-of-speech **tagsets** have many more word classes, like the 45 tags used by the Penn Treebank (Marcus et al., 1993).

tagset

POS

Parts-of-speech (also known as **POS**, **word classes**, or **syntactic categories**) are useful because of the large amount of information they give about a word and its neighbors. Knowing whether a word is a **noun** or a **verb** tells us a lot about likely neighboring words (nouns are preceded by determiners and adjectives, verbs by nouns) and about the syntactic structure around the word (nouns are generally part of noun phrases), which makes part-of-speech tagging an important component of syntactic parsing (Chapter 12). Parts of speech are useful features for finding **named**

entities like people or organizations in text and other **information extraction** tasks (Chapter 20). Parts-of-speech influence the possible morphological affixes and so can influence stemming for informational retrieval, and can help in summarization for improving the selection of nouns or other important words from a document. A word's part of speech is important for producing pronunciations in speech synthesis and recognition. The word *content*, for example, is pronounced *CONtent* when it is a noun and *conTENT* when it is an adjective (Chapter 32).

part-of-speech tagging

This chapter focuses on computational methods for assigning parts-of-speech to words, **part-of-speech tagging**. After summarizing English word classes and the standard Penn tagset, we introduce two algorithms for tagging: the Hidden Markov Model (HMM) and the Maximum Entropy Markov Model (MEMM).

10.1 (Mostly) English Word Classes

Until now we have been using part-of-speech terms like **noun** and **verb** rather freely. In this section we give a more complete definition of these and other classes. While word classes do have semantic tendencies—adjectives, for example, often describe *properties* and nouns *people*— parts-of-speech are traditionally defined instead based on syntactic and morphological function, grouping words that have similar neighboring words (their **distributional** properties) or take similar affixes (their morphological properties).

closed class
open class

Parts-of-speech can be divided into two broad supercategories: **closed class** types and **open class** types. Closed classes are those with relatively fixed membership, such as prepositions—new prepositions are rarely coined. By contrast, nouns and verbs are open classes—new nouns and verbs like *iPhone* or *to fax* are continually being created or borrowed. Any given speaker or corpus may have different open class words, but all speakers of a language, and sufficiently large corpora, likely share the set of closed class words. Closed class words are generally **function words** like *of*, *it*, *and*, or *you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

function word

Four major open classes occur in the languages of the world: **nouns**, **verbs**, **adjectives**, and **adverbs**. English has all four, although not every language does. The syntactic class **noun** includes the words for most people, places, or things, but others as well. Nouns include concrete terms like *ship* and *chair*, abstractions like *bandwidth* and *relationship*, and verb-like terms like *pacing* as in *His pacing to and fro became quite annoying*. What defines a noun in English, then, are things like its ability to occur with determiners (*a goat, its bandwidth, Plato's Republic*), to take possessives (*IBM's annual revenue*), and for most but not all nouns to occur in the plural form (*goats, abaci*).

noun

proper noun

Open class nouns fall into two classes. **Proper nouns**, like *Regina*, *Colorado*, and *IBM*, are names of specific persons or entities. In English, they generally aren't preceded by articles (e.g., *the book is upstairs*, but *Regina is upstairs*). In written English, proper nouns are usually capitalized. The other class, **common nouns** are divided in many languages, including English, into **count nouns** and **mass nouns**. Count nouns allow grammatical enumeration, occurring in both the singular and plural (*goat/goats, relationship/relationships*) and they can be counted (*one goat, two goats*). Mass nouns are used when something is conceptualized as a homogeneous group. So words like *snow, salt*, and *communism* are not counted (i.e., **two snows* or **two communisms*). Mass nouns can also appear without articles where singular

common noun
count noun
mass noun

count nouns cannot (*Snow is white* but not **Goat is white*).

verb

The **verb** class includes most of the words referring to actions and processes, including main verbs like *draw*, *provide*, and *go*. English verbs have inflections (non-third-person-sg (*eat*), third-person-sg (*eats*), progressive (*eating*), past participle (*eaten*)). While many researchers believe that all human languages have the categories of noun and verb, others have argued that some languages, such as Riau Indonesian and Tongan, don't even make this distinction (Broschart 1997; Evans 2000; Gil 2000) .

adjective

The third open class English form is **adjectives**, a class that includes many terms for properties or qualities. Most languages have adjectives for the concepts of color (*white*, *black*), age (*old*, *young*), and value (*good*, *bad*), but there are languages without adjectives. In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective "beautiful" acts in Korean like a verb meaning "to be beautiful".

adverb

The final open class form, **adverbs**, is rather a hodge-podge, both semantically and formally. In the following sentence from Schachter (1985) all the italicized words are adverbs:

Unfortunately, John walked *home extremely slowly yesterday*

locative
degree
manner
temporal

What coherence the class has semantically may be solely that each of these words can be viewed as modifying something (often verbs, hence the name "adverb", but also other adverbs and entire verb phrases). **Directional adverbs** or **locative adverbs** (*home*, *here*, *downhill*) specify the direction or location of some action; **degree adverbs** (*extremely*, *very*, *somewhat*) specify the extent of some action, process, or property; **manner adverbs** (*slowly*, *slinkily*, *delicately*) describe the manner of some action or process; and **temporal adverbs** describe the time that some action or event took place (*yesterday*, *Monday*). Because of the heterogeneous nature of this class, some adverbs (e.g., temporal adverbs like *Monday*) are tagged in some tagging schemes as nouns.

The closed classes differ more from language to language than do the open classes. Some of the important closed classes in English include:

prepositions: on, under, over, near, by, at, from, to, with
determiners: a, an, the
pronouns: she, who, I, others
conjunctions: and, but, or, as, if, when
auxiliary verbs: can, may, should, are
particles: up, down, on, off, in, out, at, by
numerals: one, two, three, first, second, third

preposition

Prepositions occur before noun phrases. Semantically they often indicate spatial or temporal relations, whether literal (*on it*, *before then*, *by the house*) or metaphorical (*on time*, *with gusto*, *beside herself*), but often indicate other relations as well, like marking the agent in (*Hamlet was written* $\underline{\textit{by}}$ *Shakespeare*,

particle

A **particle** resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren't quite the same as the prepositions they resemble, as in the particle *over* in *she turned the paper over*.

phrasal verb

When a verb and a particle behave as a single syntactic and/or semantic unit, we call the combination a **phrasal verb**. Phrasal verbs cause widespread problems with natural language processing because they often behave as a semantic unit with a **non-compositional** meaning— one that is not predictable from the distinct meanings of the verb and the particle. Thus, *turn down* means something like 'reject', *rule out* means 'eliminate', *find out* is 'discover', and *go on* is 'continue'.

determiner
article

A closed class that occurs with nouns, often marking the beginning of a noun phrase, is the **determiner**. One small subtype of determiners is the **article**: English has three articles: *a*, *an*, and *the*. Other determiners include *this* and *that* (*this chapter*, *that page*). *A* and *an* mark a noun phrase as indefinite, while *the* can mark it as definite; definiteness is a discourse property (Chapter 23). Articles are quite frequent in English; indeed, *the* is the most frequently occurring word in most corpora of written English, and *a* and *an* are generally right behind.

conjunctions

Conjunctions join two phrases, clauses, or sentences. Coordinating conjunctions like *and*, *or*, and *but* join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, *that* in *"I thought that you might like some milk"* is a subordinating conjunction that links the main clause *I thought* with the subordinate clause *you might like some milk*. This clause is called subordinate because this entire clause is the "content" of the main verb *thought*. Subordinating conjunctions like *that* which link a verb to its argument in this way are also called **complementizers**.

complementizer
pronoun
personal
possessive
wh

Pronouns are forms that often act as a kind of shorthand for referring to some noun phrase or entity or event. **Personal pronouns** refer to persons or entities (*you*, *she*, *I*, *it*, *me*, etc.). **Possessive pronouns** are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (*my, your, his, her, its, one's, our, their*). **Wh-pronouns** (*what, who, whom, whoever*) are used in certain question forms, or may also act as complementizers (*Frida, who married Diego...*).

auxiliary

A closed class subtype of English verbs are the **auxiliary** verbs. Cross-linguistically, auxiliaries mark certain semantic features of a main verb, including whether an action takes place in the present, past, or future (tense), whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood).

copula
modal

English auxiliaries include the **copula** verb *be*, the two verbs *do* and *have*, along with their inflected forms, as well as a class of **modal verbs**. *Be* is called a copula because it connects subjects with certain kinds of predicate nominals and adjectives (*He is a duck*). The verb *have* is used, for example, to mark the perfect tenses (*I have gone*, *I had gone*), and *be* is used as part of the passive (*We were robbed*) or progressive (*We are leaving*) constructions. The modals are used to mark the mood associated with the event or action depicted by the main verb: *can* indicates ability or possibility, *may* indicates permission or possibility, *must* indicates necessity. In addition to the perfect *have* mentioned above, there is a modal verb *have* (e.g., *I have to go*), which is common in spoken English.

interjection
negative

English also has many words of more or less unique function, including **interjections** (*oh, hey, alas, uh, um*), **negatives** (*no, not*), **politeness markers** (*please, thank you*), **greetings** (*hello, goodbye*), and the existential **there** (*there are two on the table*) among others. These classes may be distinguished or lumped together as interjections or adverbs depending on the purpose of the labeling.

10.2 The Penn Treebank Part-of-Speech Tagset

While there are many lists of parts-of-speech, most modern language processing on English uses the 45-tag Penn Treebank tagset (Marcus et al., 1993), shown in Fig. 10.1. This tagset has been used to label a wide variety of corpora, including the Brown corpus, the *Wall Street Journal* corpus, and the Switchboard corpus.

Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	*and, but, or*	SYM	symbol	+,%, &
CD	cardinal number	*one, two*	TO	"to"	*to*
DT	determiner	*a, the*	UH	interjection	*ah, oops*
EX	existential 'there'	*there*	VB	verb base form	*eat*
FW	foreign word	*mea culpa*	VBD	verb past tense	*ate*
IN	preposition/sub-conj	*of, in, by*	VBG	verb gerund	*eating*
JJ	adjective	*yellow*	VBN	verb past participle	*eaten*
JJR	adj., comparative	*bigger*	VBP	verb non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	VBZ	verb 3sg pres	*eats*
LS	list item marker	*1, 2, One*	WDT	wh-determiner	*which, that*
MD	modal	*can, should*	WP	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama*	WP$	possessive wh-	*whose*
NNS	noun, plural	*llamas*	WRB	wh-adverb	*how, where*
NNP	proper noun, sing.	*IBM*	$	dollar sign	$
NNPS	proper noun, plural	*Carolinas*	#	pound sign	#
PDT	predeterminer	*all, both*	"	left quote	' or "
POS	possessive ending	*'s*	"	right quote	' or "
PRP	personal pronoun	*I, you, he*	(	left parenthesis	[, (, {, <
PRP$	possessive pronoun	*your, one's*	)	right parenthesis	],), }, >
RB	adverb	*quickly, never*	,	comma	,
RBR	adverb, comparative	*faster*	.	sentence-final punc	. ! ?
RBS	adverb, superlative	*fastest*	:	mid-sentence punc	: ; ... – -
RP	particle	*up, off*			

Figure 10.1 Penn Treebank part-of-speech tags (including punctuation).

Parts-of-speech are generally represented by placing the tag after each word, delimited by a slash, as in the following examples:

(10.1) The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS ./.

(10.2) **There/EX** are/VBP 70/CD children/NNS **there/RB**

(10.3) Preliminary/JJ findings/NNS were/VBD **reported/VBN** in/IN today/NN **'s/POS** New/NNP England/NNP Journal/NNP of/IN Medicine/NNP ./.

Example (10.1) shows the determiners *the* and *a*, the adjectives *grand* and *other*, the common nouns *jury*, *number*, and *topics*, and the past tense verb *commented*. Example (10.2) shows the use of the EX tag to mark the existential *there* construction in English, and, for comparison, another use of *there* which is tagged as an adverb (RB). Example (10.3) shows the segmentation of the possessive morpheme *'s* a passive construction, 'were reported', in which *reported* is marked as a past participle (VBN). Note that since *New England Journal of Medicine* is a proper noun, the Treebank tagging chooses to mark each noun in it separately as NNP, including *journal* and *medicine*, which might otherwise be labeled as common nouns (NN).

Corpora labeled with parts-of-speech like the Treebank corpora are crucial training (and testing) sets for statistical tagging algorithms. Three main tagged corpora are consistently used for training and testing part-of-speech taggers for English (see Section 10.7 for other languages). The **Brown** corpus is a million words of samples from 500 written texts from different genres published in the United States in 1961. The **WSJ** corpus contains a million words published in the Wall Street Journal in 1989. The **Switchboard** corpus consists of 2 million words of telephone conversations collected in 1990-1991. The corpora were created by running an automatic

Brown

WSJ

Switchboard

part-of-speech tagger on the texts and then human annotators hand-corrected each tag.

There are some minor differences in the tagsets used by the corpora. For example in the WSJ and Brown corpora, the single Penn tag TO is used for both the infinitive *to* (*I like to race*) and the preposition *to* (*go to the store*), while in the Switchboard corpus the tag TO is reserved for the infinitive use of *to*, while the preposition use is tagged IN:

Well/UH ,/, I/PRP ,/, I/PRP want/VBP **to/TO** go/VB **to/IN** a/DT restaurant/NN

Finally, there are some idiosyncracies inherent in any tagset. For example, because the Penn 45 tags were collapsed from a larger 87-tag tagset, the **original Brown tagset**, some potential useful distinctions were lost. The Penn tagset was designed for a treebank in which sentences were parsed, and so it leaves off syntactic information recoverable from the parse tree. Thus for example the Penn tag IN is used for both subordinating conjunctions like *if, when, unless, after*:

after/IN spending/VBG a/DT day/NN at/IN the/DT beach/NN

and prepositions like *in, on, after*:

after/IN sunrise/NN

Tagging algorithms assume that words have been tokenized before tagging. The Penn Treebank and the British National Corpus split contractions and the *'s*-genitive from their stems:

would/MD n't/RB
children/NNS 's/POS

Indeed, the special Treebank tag POS is used only for the morpheme *'s*, which must be segmented off during tokenization.

Another tokenization issue concerns multipart words. The Treebank tagset assumes that tokenization of words like *New York* is done at whitespace. The phrase *a New York City firm* is tagged in Treebank notation as five separate words: *a/DT New/NNP York/NNP City/NNP firm/NN*. The C5 tagset for the British National Corpus, by contrast, allow prepositions like "*in terms of*" to be treated as a single word by adding numbers to each tag, as in *in/II31 terms/II32 of/II33*.

10.3 Part-of-Speech Tagging

tagging

Part-of-speech tagging (**tagging** for short) is the process of assigning a part-of-speech marker to each word in an input text. Because tags are generally also applied to punctuation, **tokenization** is usually performed before, or as part of, the tagging process: separating commas, quotation marks, etc., from words and disambiguating end-of-sentence punctuation (period, question mark, etc.) from part-of-word punctuation (such as in abbreviations like *e.g.* and *etc.*)

The input to a tagging algorithm is a sequence of words and a tagset, and the output is a sequence of tags, a single best tag for each word as shown in the examples on the previous pages.

ambiguous

Tagging is a **disambiguation** task; words are **ambiguous** —have more than one possible part-of-speech— and the goal is to find the correct tag for the situation. For example, the word *book* can be a verb (*book that flight*) or a noun (as in *hand me that book*.

That can be a determiner (*Does that flight serve dinner*) or a complementizer (*I thought that your flight was earlier*). The problem of POS-tagging is to **resolve** these ambiguities, choosing the proper tag for the context. Part-of-speech tagging is thus one of the many **disambiguation** tasks in language processing.

resolution

disambiguation

How hard is the tagging problem? And how common is tag ambiguity? Fig. 10.2 shows the answer for the Brown and WSJ corpora tagged using the 45-tag Penn tagset. Most word types (80-86%) are unambiguous; that is, they have only a single tag (*Janet* is always NNP, *funniest* JJS, and *hesitantly* RB). But the ambiguous words, although accounting for only 14-15% of the vocabulary, are some of the most common words of English, and hence 55-67% of word tokens in running text are ambiguous. Note the large differences across the two genres, especially in token frequency. Tags in the WSJ corpus are less ambiguous, presumably because this newspaper's specific focus on financial news leads to a more limited distribution of word usages than the more general texts combined into the Brown corpus.

Types:		**WSJ**		**Brown**	
Unambiguous	(1 tag)	44,432	(**86%**)	45,799	(**85%**)
Ambiguous	(2+ tags)	7,025	(**14%**)	8,050	(**15%**)
Tokens:					
Unambiguous	(1 tag)	577,421	(**45%**)	384,349	(**33%**)
Ambiguous	(2+ tags)	711,780	(**55%**)	786,646	(**67%**)

Figure 10.2 The amount of tag ambiguity for word types in the Brown and WSJ corpora, from the Treebank-3 (45-tag) tagging. These statistics include punctuation as words, and assume words are kept in their original case.

Some of the most ambiguous frequent words are *that*, *back*, *down*, *put* and *set*; here are some examples of the 6 different parts-of-speech for the word *back*:

earnings growth took a **back/JJ** seat
a small building in the **back/NN**
a clear majority of senators **back/VBP** the bill
Dave began to **back/VB** toward the door
enable the country to buy **back/RP** about debt
I was twenty-one **back/RB** then

Still, even many of the ambiguous tokens are easy to disambiguate. This is because the different tags associated with a word are not equally likely. For example, *a* can be a determiner or the letter *a* (perhaps as part of an acronym or an initial). But the determiner sense of *a* is much more likely. This idea suggests a simplistic **baseline** algorithm for part of speech tagging: given an ambiguous word, choose the tag which is **most frequent** in the training corpus. This is a key concept:

Most Frequent Class Baseline: Always compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

accuracy

How good is this baseline? A standard way to measure the performance of part-of-speech taggers is **accuracy**: the percentage of tags correctly labeled on a human-labeled test set. One commonly used test set is sections 22-24 of the WSJ corpus. If we train on the rest of the WSJ corpus and test on that test set, the most-frequent-tag baseline achieves an accuracy of 92.34%.

By contrast, the state of the art in part-of-speech tagging on this dataset is around 97% tag accuracy, a performance that is achievable by a number of statistical algorithms including HMMs, MEMMs and other log-linear models, perceptrons, and probably also rule-based systems—see the discussion at the end of the chapter. See Section 10.7 on other languages and genres.

10.4 HMM Part-of-Speech Tagging

In this section we introduce the use of the Hidden Markov Model for part-of-speech tagging. The HMM defined in the previous chapter was quite powerful, including a learning algorithm— the Baum-Welch (EM) algorithm—that can be given unlabeled data and find the best mapping of labels to observations. However when we apply HMM to part-of-speech tagging we generally don't use the Baum-Welch algorithm for learning the HMM parameters. Instead HMMs for part-of-speech tagging are trained on a fully labeled dataset—a set of sentences with each word annotated with a part-of-speech tag—setting parameters by maximum likelihood estimates on this training data.

Thus the only algorithm we will need from the previous chapter is the **Viterbi** algorithm for decoding, and we will also need to see how to set the parameters from training data.

10.4.1 The basic equation of HMM Tagging

Let's begin with a quick reminder of the intuition of HMM decoding. The goal of HMM decoding is to choose the tag sequence that is most probable given the observation sequence of n words w_1^n:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \tag{10.4}$$

by using Bayes' rule to instead compute:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} \frac{P(w_1^n|t_1^n)P(t_1^n)}{P(w_1^n)} \tag{10.5}$$

Furthermore, we simplify Eq. 10.5 by dropping the denominator $P(w_1^n)$:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(w_1^n|t_1^n)P(t_1^n) \tag{10.6}$$

HMM taggers make two further simplifying assumptions. The first is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1^n|t_1^n) \approx \prod_{i=1}^{n} P(w_i|t_i) \tag{10.7}$$

The second assumption, the **bigram** assumption, is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence;

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{10.8}$$

Plugging the simplifying assumptions from Eq. 10.7 and Eq. 10.8 into Eq. 10.6 results in the following equation for the most probable tag sequence from a bigram tagger, which as we will soon see, correspond to the **emission probability** and **transition probability** from the HMM of Chapter 9.

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} \overbrace{P(w_i|t_i)}^{\text{emission}} \; \overbrace{P(t_i|t_{i-1})}^{\text{transition}} \tag{10.9}$$

10.4.2 Estimating probabilities

Let's walk through an example, seeing how these probabilities are estimated and used in a sample tagging task, before we return to the Viterbi algorithm.

In HMM tagging, rather than using the full power of HMM EM learning, the probabilities are estimated just by counting on a tagged training corpus. For this example we'll use the tagged WSJ corpus. The tag transition probabilities $P(t_i|t_{i-1})$ represent the probability of a tag given the previous tag. For example, modal verbs like *will* are very likely to be followed by a verb in the base form, a VB, like *race*, so we expect this probability to be high. The maximum likelihood estimate of a transition probability is computed by counting, out of the times we see the first tag in a labeled corpus, how often the first tag is followed by the second

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})} \tag{10.10}$$

In the WSJ corpus, for example, MD occurs 13124 times of which it is followed by VB 10471, for an MLE estimate of

$$P(VB|MD) = \frac{C(MD,VB)}{C(MD)} = \frac{10471}{13124} = .80 \tag{10.11}$$

The emission probabilities, $P(w_i|t_i)$, represent the probability, given a tag (say MD), that it will be associated with a given word (say *will*). The MLE of the emission probability is

$$P(w_i|t_i) = \frac{C(t_i,w_i)}{C(t_i)} \tag{10.12}$$

Of the 13124 occurrences of MD in the WSJ corpus, it is associated with *will* 4046 times:

$$P(will|MD) = \frac{C(MD,will)}{C(MD)} = \frac{4046}{13124} = .31 \tag{10.13}$$

For those readers who are new to Bayesian modeling, note that this likelihood term is not asking "which is the most likely tag for the word *will*?" That would be the posterior $P(\text{MD}|\text{will})$. Instead, $P(\text{will}|\text{MD})$ answers the slightly counterintuitive question "If we were going to generate a MD, how likely is it that this modal would be *will*?"

The two kinds of probabilities from Eq. 10.9, the transition (prior) probabilities like $P(VB|MD)$ and the emission (likelihood) probabilities like $P(will|MD)$, correspond to the A transition probabilities, and B observation likelihoods of the HMM. Figure 10.3 illustrates some of the the A transition probabilities for three states in an HMM part-of-speech tagger; the full tagger would have one state for each tag.

Figure 10.4 shows another view of these three states from an HMM tagger, focusing on the word likelihoods B. Each hidden state is associated with a vector of likelihoods for each observation word.

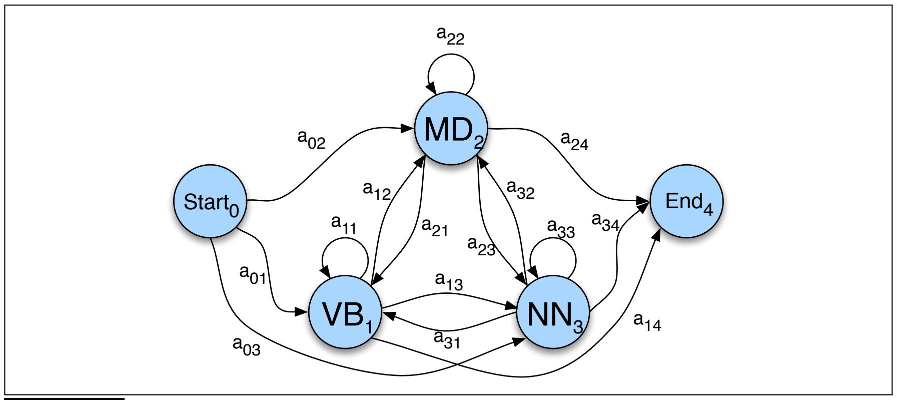

Figure 10.3 A piece of the Markov chain corresponding to the hidden states of the HMM. The *A* transition probabilities are used to compute the prior probability.

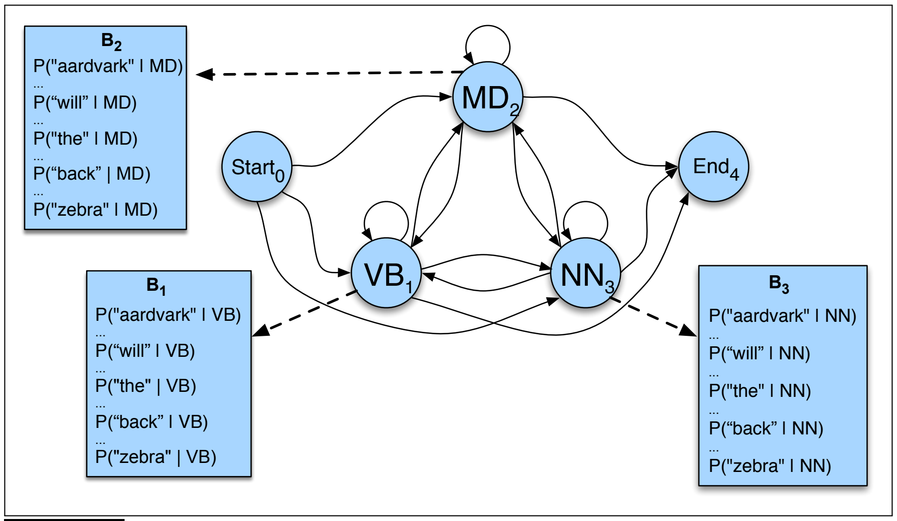

Figure 10.4 Some of the *B* observation likelihoods for the HMM in the previous figure. Each state (except the non-emitting start and end states) is associated with a vector of probabilities, one likelihood for each possible observation word.

10.4.3 Working through an example

Let's now work through an example of computing the best sequence of tags that corresponds to the following sequence of words

(10.14) Janet will back the bill

The correct series of tags is:

(10.15) Janet/NNP will/MD back/VB the/DT bill/NN

Let the HMM be defined by the two tables in Fig. 10.5 and Fig. 10.6.

Figure 10.5 lists the a_{ij} probabilities for transitioning between the hidden states (part-of-speech tags).

Figure 10.6 expresses the $b_i(o_t)$ probabilities, the *observation* likelihoods of words given tags. This table is (slightly simplified) from counts in the WSJ corpus. So the word *Janet* only appears as an NNP, *back* has 4 possible parts of speech, and the word *the* can appear as a determiner or as an NNP (in titles like "Somewhere Over the Rainbow" all words are tagged as NNP).

	NNP	**MD**	**VB**	**JJ**	**NN**	**RB**	**DT**
$<s>$	0.2767	0.0006	0.0031	0.0453	0.0449	0.0510	0.2026
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	0.0090	0.0025
MD	0.0008	0.0002	0.7968	0.0005	0.0008	0.1698	0.0041
VB	0.0322	0.0005	0.0050	0.0837	0.0615	0.0514	0.2231
JJ	0.0366	0.0004	0.0001	0.0733	0.4509	0.0036	0.0036
NN	0.0096	0.0176	0.0014	0.0086	0.1216	0.0177	0.0068
RB	0.0068	0.0102	0.1011	0.1012	0.0120	0.0728	0.0479
DT	0.1147	0.0021	0.0002	0.2157	0.4744	0.0102	0.0017

Figure 10.5 The A transition probabilities $P(t_i|t_{i-1})$ computed from the WSJ corpus without smoothing. Rows are labeled with the conditioning event; thus $P(VB|MD)$ is 0.7968.

	Janet	**will**	**back**	**the**	**bill**
NNP	0.000032	0	0	0.000048	0
MD	0	0.308431	0	0	0
VB	0	0.000028	0.000672	0	0.000028
JJ	0	0	0.000340	0.000097	0
NN	0	0.000200	0.000223	0.000006	0.002337
RB	0	0	0.010446	0	0
DT	0	0	0	0.506099	0

Figure 10.6 Observation likelihoods B computed from the WSJ corpus without smoothing.

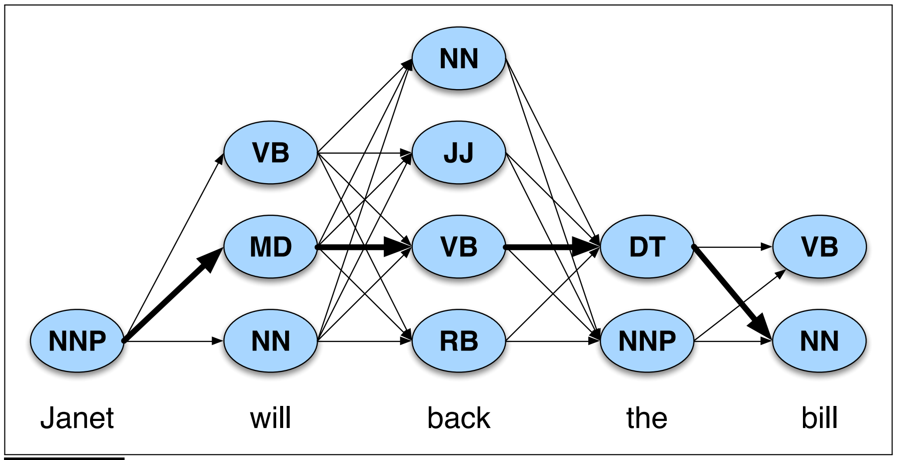

Figure 10.7 A schematic of the tagging task for the sample sentence, showing the ambiguities for each word and the correct tag sequence as the highlighted path through the hidden states.

Figure 10.7 shows a schematic of the possible tags for each word and the correct final tag sequence.

For the reader's convenience Fig. 10.8 repeats the pseudocode for the Viterbi algorithm from Chapter 9. The Viterbi algorithm sets up a probability matrix, with one column for each observation t and one row for each state in the state graph. Each column thus has a cell for each state q_i in the single combined automaton for the four words.

The algorithm first creates $N = 5$ state columns, the first for the observation of the first word *Janet*, the second for *will*, and so on. We begin in the first column by setting the Viterbi value in each cell to the product of the transition probability (into it from the start state) and the observation probability (of the first word); the reader should find this in Fig. 10.9.

```
function VITERBI(observations of len T,state-graph of len N) returns best-path

  create a path probability matrix viterbi[N+2,T]
  for each state s from 1 to N do                          ; initialization step
      $viterbi[s,1] \leftarrow a_{0,s} * b_s(o_1)$
      $backpointer[s,1] \leftarrow 0$
  for each time step t from 2 to T do                      ; recursion step
    for each state s from 1 to N do
      $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
      $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s}$
  $viterbi[q_F,T] \leftarrow \max_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$          ; termination step
  $backpointer[q_F,T] \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$     ; termination step
  return the backtrace path by following backpointers to states back in time from
backpointer[$q_F$, $T$]
```

Figure 10.8 Viterbi algorithm for finding optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence. Note that states 0 and q_F are non-emitting.

Then we move on, column by column; for every state in column 1, we compute the probability of moving into each state in column 2, and so on. For each state q_j at time t, we compute the value *viterbi*$[s,t]$ by taking the maximum over the extensions of all the paths that lead to the current cell, using the following equation:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t) \tag{10.16}$$

Recall from Chapter 9 that the three factors that are multiplied in Eq. 10.16 for extending the previous paths to compute the Viterbi probability at time t are

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

In Fig. 10.9, each cell of the trellis in the column for the word *Janet* is computed by multiplying the previous probability at the start state (1.0), the transition probability from the start state to the tag for that cell, and the observation likelihood of the word *Janet* given the tag for that cell. Most of the cells in the column are zero since the word *Janet* cannot be any of those tags. Next, each cell in the *will* column gets updated with the maximum probability path from the previous column. We have shown the values for the MD, VB, and NN cells. Each cell gets the max of the 7 values from the previous column, multiplied by the appropriate transition probability; as it happens in this case, most of them are zero from the previous column. The remaining value is multiplied by the relevant transition probability, and the (trivial) max is taken. In this case the final value, .0000002772, comes from the NNP state at the previous column. The reader should fill in the rest of the trellis in Fig. 10.9 and backtrace to reconstruct the correct state sequence NNP MD VB DT NN. (Exercise 10.**??**).

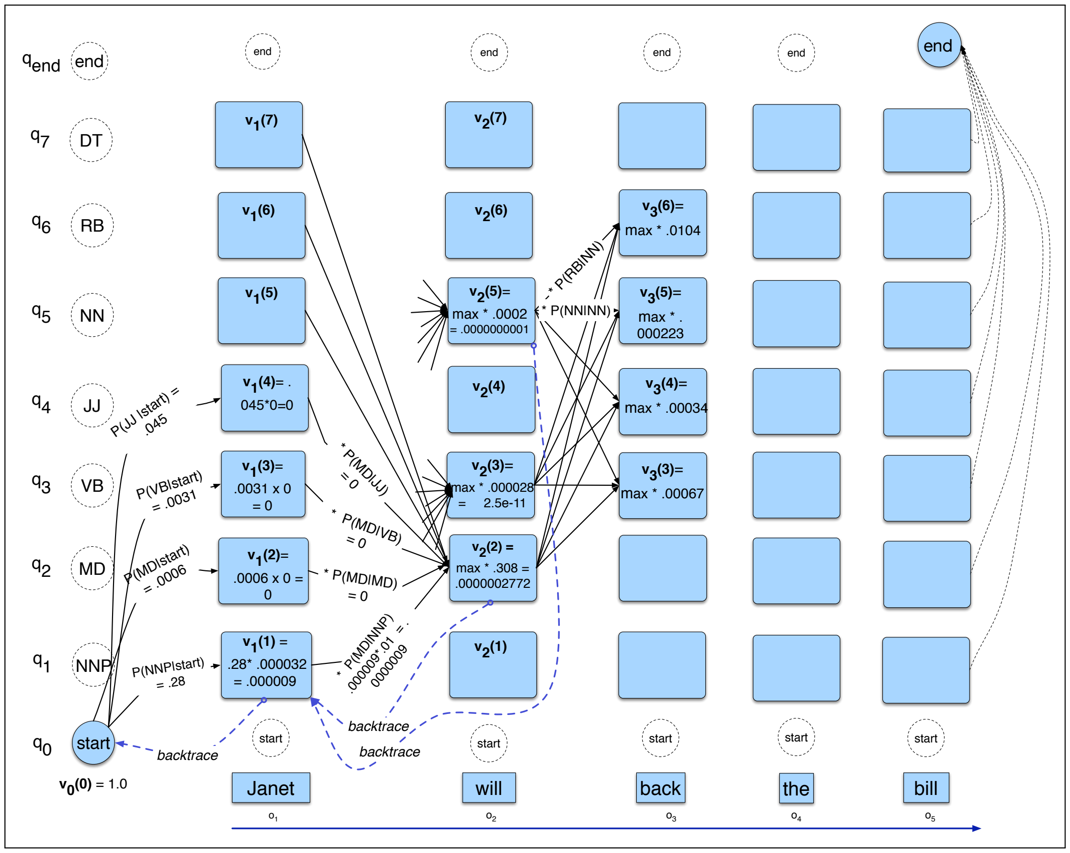

Figure 10.9 The first few entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. We have only filled out columns 1 and 2; to avoid clutter most cells with value 0 are left empty. The rest is left as an exercise for the reader. After the cells are filled in, backtracing from the *end* state, we should be able to reconstruct the correct state sequence NNP MD VB DT NN.

10.4.4 Extending the HMM Algorithm to Trigrams

Practical HMM taggers have a number of extensions of this simple model. One important missing feature is a wider tag context. In the tagger described above the probability of a tag depends only on the previous tag:

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{10.17}$$

In practice we use more of the history, letting the probability of a tag depend on the two previous tags:

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1},t_{i-2}) \tag{10.18}$$

Extending the algorithm from bigram to trigram taggers gives a small (perhaps a half point) increase in performance, but conditioning on two previous tags instead of one requires a significant change to the Viterbi algorithm. For each cell, instead of taking a max over transitions from each cell in the previous column, we have to take a max over paths through the cells in the previous two columns, thus considering N^2 rather than N hidden states at every observation.

In addition to increasing the context window, state-of-the-art HMM taggers like Brants (2000) have a number of other advanced features. One is to let the tagger know the location of the end of the sentence by adding dependence on an end-of-sequence marker for t_{n+1}. This gives the following equation for part-of-speech tagging:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \left[\prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1},t_{i-2})\right] P(t_{n+1}|t_n) \qquad (10.19)$$

In tagging any sentence with Eq. 10.19, three of the tags used in the context will fall off the edge of the sentence, and hence will not match regular words. These tags, t_{-1}, t_0, and t_{n+1}, can all be set to be a single special 'sentence boundary' tag that is added to the tagset, which assumes sentences boundaries have already been marked.

One problem with trigram taggers as instantiated in Eq. 10.19 is data sparsity. Any particular sequence of tags t_{i-2},t_{i-1},t_i that occurs in the test set may simply never have occurred in the training set. That means we cannot compute the tag trigram probability just by the maximum likelihood estimate from counts, following Eq. 10.20:

$$P(t_i|t_{i-1},t_{i-2}) = \frac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-2},t_{i-1})} \qquad (10.20)$$

Just as we saw with language modeling, many of these counts will be zero in any training set, and we will incorrectly predict that a given tag sequence will never occur! What we need is a way to estimate $P(t_i|t_{i-1},t_{i-2})$ even if the sequence t_{i-2},t_{i-1},t_i never occurs in the training data.

The standard approach to solving this problem is the same interpolation idea we saw in language modeling: estimate the probability by combining more robust, but weaker estimators. For example, if we've never seen the tag sequence PRP VB TO, and so can't compute $P(\text{TO}|\text{PRP,VB})$ from this frequency, we still could rely on the bigram probability $P(\text{TO}|\text{VB})$, or even the unigram probability $P(\text{TO})$. The maximum likelihood estimation of each of these probabilities can be computed from a corpus with the following counts:

$$\text{Trigrams} \quad \hat{P}(t_i|t_{i-1},t_{i-2}) = \frac{C(t_{i-2},t_{i-1},t_i)}{C(t_{i-2},t_{i-1})} \qquad (10.21)$$

$$\text{Bigrams} \quad \hat{P}(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})} \qquad (10.22)$$

$$\text{Unigrams} \quad \hat{P}(t_i) = \frac{C(t_i)}{N} \qquad (10.23)$$

The standard way to combine these three estimators to estimate the trigram probability $P(t_i|t_{i-1},t_{i-2})$? is via linear interpolation. We estimate the probability $P(t_i|t_{i-1}t_{i-2})$ by a weighted sum of the unigram, bigram, and trigram probabilities:

$$P(t_i|t_{i-1}t_{i-2}) = \lambda_3\hat{P}(t_i|t_{i-1}t_{i-2}) + \lambda_2\hat{P}(t_i|t_{i-1}) + \lambda_1\hat{P}(t_i) \qquad (10.24)$$

deleted interpolation

We require $\lambda_1 + \lambda_2 + \lambda_3 = 1$, ensuring that the resulting P is a probability distribution. These λs are generally set by an algorithm called **deleted interpolation**

(Jelinek and Mercer, 1980): we successively delete each trigram from the training corpus and choose the λs so as to maximize the likelihood of the rest of the corpus. The deletion helps to set the λs in such a way as to generalize to unseen data and not overfit the training corpus. Figure 10.10 gives a deleted interpolation algorithm for tag trigrams.

```
function DELETED-INTERPOLATION(corpus) returns $\lambda_1, \lambda_2, \lambda_3$

  $\lambda_1 \leftarrow 0$
  $\lambda_2 \leftarrow 0$
  $\lambda_3 \leftarrow 0$
  foreach trigram $t_1, t_2, t_3$ with $C(t_1, t_2, t_3) > 0$
    depending on the maximum of the following three values
      case $\frac{C(t_1,t_2,t_3)-1}{C(t_1,t_2)-1}$: increment $\lambda_3$ by $C(t_1, t_2, t_3)$
      case $\frac{C(t_2,t_3)-1}{C(t_2)-1}$: increment $\lambda_2$ by $C(t_1, t_2, t_3)$
      case $\frac{C(t_3)-1}{N-1}$: increment $\lambda_1$ by $C(t_1, t_2, t_3)$
    end
  end
  normalize $\lambda_1, \lambda_2, \lambda_3$
  return $\lambda_1, \lambda_2, \lambda_3$
```

Figure 10.10 The deleted interpolation algorithm for setting the weights for combining unigram, bigram, and trigram tag probabilities. If the denominator is 0 for any case, we define the result of that case to be 0. N is the total number of tokens in the corpus. After Brants (2000).

10.4.5 Unknown Words

words people
never use —
could be
only I
know them
Ishikawa Takuboku 1885–1912

unknown words

To achieve high accuracy with part-of-speech taggers, it is also important to have a good model for dealing with **unknown words**. Proper names and acronyms are created very often, and even new common nouns and verbs enter the language at a surprising rate. One useful feature for distinguishing parts of speech is wordshape: words starting with capital letters are likely to be proper nouns (NNP).

But the strongest source of information for guessing the part-of-speech of unknown words is morphology. Words that end in *-s* are likely to be plural nouns (NNS), words ending with *-ed* tend to be past participles (VBN), words ending with *-able* tend to be adjectives (JJ), and so on. One way to take advantage of this is to store for each final letter sequence (for simplicity referred to as word *suffixes*) the statistics of which tag they were associated with in training. The method of Samuelsson (1993) and Brants (2000), for example, considers suffixes of up to ten letters, computing for each suffix of length i the probability of the tag t_i given the suffix letters:

$$P(t_i|l_{n-i+1} \dots l_n) \tag{10.25}$$

They use back-off to smooth these probabilities with successively shorter and shorter suffixes. To capture the fact that unknown words are unlikely to be closed-class words like prepositions, we can compute suffix probabilities only from the training set for words whose frequency in the training set is ≤ 10, or alternately can train suffix probabilities only on open-class words. Separate suffix tries are kept for capitalized and uncapitalized words.

Finally, because Eq. 10.25 gives a posterior estimate $p(t_i|w_i)$, we can compute the likelihood $p(w_i|t_i)$ that HMMs require by using Bayesian inversion (i.e., using Bayes rule and computation of the two priors $P(t_i)$ and $P(t_i|l_{n-i+1} \dots l_n)$).

In addition to using capitalization information for unknown words, Brants (2000) also uses capitalization for known words by adding a capitalization feature to each tag. Thus, instead of computing $P(t_i|t_{i-1},t_{i-2})$ as in Eq. 10.21, the algorithm computes the probability $P(t_i,c_i|t_{i-1},c_{i-1},t_{i-2},c_{i-2})$. This is equivalent to having a capitalized and uncapitalized version of each tag, essentially doubling the size of the tagset.

Combining all these features, a state-of-the-art trigram HMM like that of Brants (2000) has a tagging accuracy of 96.7% on the Penn Treebank.

10.5 Maximum Entropy Markov Models

We turn now to a second sequence model, the **maximum entropy Markov model** or **MEMM**. The MEMM is a sequence model adaptation of the MaxEnt (multinomial logistic regression) classifier. Because it is based on logistic regression, the MEMM is a **discriminative sequence model**. By contrast, the HMM is a **generative sequence model**.

MEMM

discriminative

generative

Let the sequence of words be $W = w_1^n$ and the sequence of tags $T = t_1^n$. In an HMM to compute the best tag sequence that maximizes $P(T|W)$ we rely on Bayes' rule and the likelihood $P(W|T)$:

$$\begin{aligned}\hat{T} &= \operatorname*{argmax}_T P(T|W) \\ &= \operatorname*{argmax}_T P(W|T)P(T) \\ &= \operatorname*{argmax}_T \prod_i P(word_i|tag_i) \prod_i P(tag_i|tag_{i-1}) \qquad (10.26)\end{aligned}$$

In an MEMM, by contrast, we compute the posterior $P(T|W)$ directly, training it to discriminate among the possible tag sequences:

$$\begin{aligned}\hat{T} &= \operatorname*{argmax}_T P(T|W) \\ &= \operatorname*{argmax}_T \prod_i P(t_i|w_i,t_{i-1}) \qquad (10.27)\end{aligned}$$

We could do this by training a logistic regression classifier to compute the single probability $P(t_i|w_i,t_{i-1})$. Fig. 10.11 shows the intuition of the difference via the direction of the arrows; HMMs compute likelihood (observation word conditioned on tags) but MEMMs compute posterior (tags conditioned on observation words).

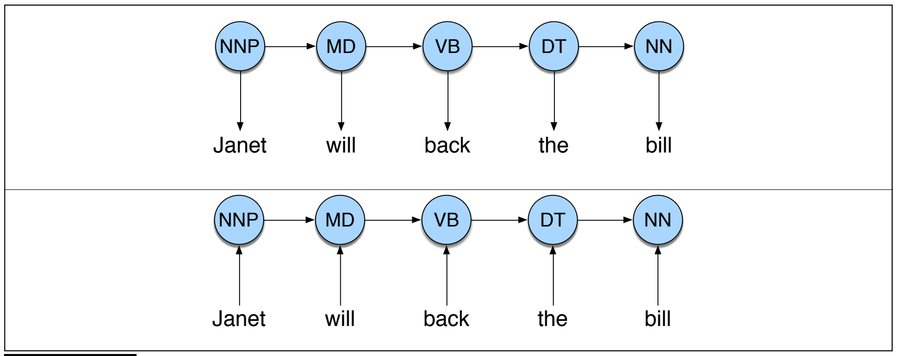

Figure 10.11 A schematic view of the HMM (top) and MEMM (bottom) representation of the probability computation for the correct sequence of tags for the *back* sentence. The HMM computes the likelihood of the observation given the hidden state, while the MEMM computes the posterior of each state, conditioned on the previous state and current observation.

10.5.1 Features in a MEMM

Oops. We lied in Eq. 10.27. We actually don't build MEMMs that condition just on w_i and t_{i-1}. In fact, an MEMM conditioned on just these two features (the observed word and the previous tag), as shown in Fig. 10.11 and Eq. 10.27 is no more accurate than the generative HMM model and in fact may be less accurate.

The reason to use a discriminative sequence model is that discriminative models make it easier to incorporate a much wider variety of features. Because in HMMs all computation is based on the two probabilities $P(\text{tag}|\text{tag})$ and $P(\text{word}|\text{tag})$, if we want to include some source of knowledge into the tagging process, we must find a way to encode the knowledge into one of these two probabilities. We saw in the previous section that it was possible to model capitalization or word endings by cleverly fitting in probabilities like $P(\text{capitalization}|\text{tag})$, $P(\text{suffix}|\text{tag})$, and so on into an HMM-style model. But each time we add a feature we have to do a lot of complicated conditioning which gets harder and harder as we have more and more such features and, as we'll see, there are lots more features we can add. Figure 10.12 shows a graphical intuition of some of these additional features.

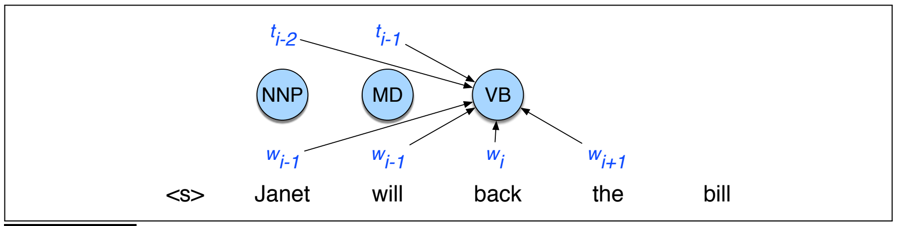

Figure 10.12 An MEMM for part-of-speech tagging showing the ability to condition on more features.

A basic MEMM part-of-speech tagger conditions on the observation word itself, neighboring words, and previous tags, and various combinations, using feature **templates** like the following:

templates

$$\begin{aligned}\langle t_i, w_{i-2}\rangle, \langle t_i, w_{i-1}\rangle, \langle t_i, w_i\rangle, \langle t_i, w_{i+1}\rangle, \langle t_i, w_{i+2}\rangle \\ \langle t_i, t_{i-1}\rangle, \langle t_i, t_{i-2}, t_{i-1}\rangle, \\ \langle t_i, t_{i-1}, w_i\rangle, \langle t_i, w_{i-1}, w_i\rangle \langle t_i, w_i, w_{i+1}\rangle,\end{aligned} \quad (10.28)$$

Recall from Chapter 7 that feature templates are used to automatically populate the set of features from every instance in the training and test set. Thus our example *Janet/NNP will/MD back/VB the/DT bill/NN*, when w_i is the word *back*, would generate the following features:

t_i = VB and w_{i-2} = Janet
t_i = VB and w_{i-1} = will
t_i = VB and w_i = back
t_i = VB and w_{i+1} = the
t_i = VB and w_{i+2} = bill
t_i = VB and t_{i-1} = MD
t_i = VB and t_{i-1} = MD and t_{i-2} = NNP
t_i = VB and w_i = back and w_{i+1} = the

Also necessary are features to deal with unknown words, expressing properties of the word's spelling or shape:

w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
w_i contains a number
w_i contains an upper-case letter
w_i contains a hyphen
w_i is all upper case
w_i's word shape
w_i's short word shape
w_i is upper case and has a digit and a dash (like *CFC-12*)
w_i is upper case and followed within 3 words by Co., Inc., etc.

word shape

Word shape features are used to represent the abstract letter pattern of the word by mapping lower-case letters to 'x', upper-case to 'X', numbers to 'd', and retaining punctuation. Thus for example I.M.F would map to X.X.X. and DC10-30 would map to XXdd-dd. A second class of shorter word shape features is also used. In these features consecutive character types are removed, so DC10-30 would be mapped to Xd-d but I.M.F would still map to X.X.X. For example the word *well-dressed* would generate the following non-zero valued feature values:

prefix(w_i) = `w`
prefix(w_i) = `we`
prefix(w_i) = `wel`
prefix(w_i) = `well`
suffix(w_i) = `ssed`
suffix(w_i) = `sed`
suffix(w_i) = `ed`
suffix(w_i) = `d`
has-hyphen(w_i)
word-shape(w_i) = `xxxx-xxxxxxx`
short-word-shape(w_i) = `x-x`

Features for known words, like the templates in Eq. 10.28, are computed for every word seen in the training set. The unknown word features can also be computed for all words in training, or only on rare training words whose frequency is below some threshold.

The result of the known-word templates and word-signature features is a very large set of features. Generally a feature cutoff is used in which features are thrown out if they have count < 5 in the training set.

Given this large set of features, the most likely sequence of tags is then computed by a MaxEnt model that combines these features of the input word w_i, its neighbors within l words w_{i-l}^{i+l}, and the previous k tags t_{i-k}^{i-1} as follows:

$$\begin{aligned}\hat{T} &= \underset{T}{\operatorname{argmax}}\, P(T|W) \\ &= \underset{T}{\operatorname{argmax}} \prod_i P(t_i|w_{i-l}^{i+l}, t_{i-k}^{i-1}) \\ &= \underset{T}{\operatorname{argmax}} \prod_i \frac{\exp\left(\sum_i w_i f_i(t_i, w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}{\sum_{t' \in \text{tagset}} \exp\left(\sum_i w_i f_i(t', w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}\end{aligned} \tag{10.29}$$

10.5.2 Decoding and Training MEMMs

We're now ready to see how to use the MaxEnt classifier to solve the decoding problem by finding the most likely sequence of tags described in Eq. 10.29.

The simplest way to turn the MaxEnt classifier into a sequence model is to build a local classifier that classifies each word left to right, making a hard classification of the first word in the sentence, then a hard decision on the the second word, and so on. This is called a **greedy** decoding algorithm, because we greedily choose the best tag for each word, as shown in Fig. 10.13.

greedy

```
function GREEDY MEMM DECODING(words W, model P) returns tag sequence T

for i = 1 to length(W)
    $\hat{t}_i = \underset{t' \in T}{\operatorname{argmax}}\ P(t' \mid w_{i-l}^{i+l}, t_{i-k}^{i-1})$
```

Figure 10.13 In greedy decoding we make a hard decision to choose the best tag left to right.

The problem with the greedy algorithm is that by making a hard decision on each word before moving on to the next word, the classifier cannot temper its decision with information from future decisions. Although greedy algorithm is very fast, and we do use it in some applications when it has sufficient accuracy, in general this hard decision causes sufficient drop in performance that we don't use it.

Viterbi

Instead we decode an MEMM with the **Viterbi** algorithm just as we did with the HMM, thus finding the sequence of part-of-speech tags that is optimal for the whole sentence.

Let's see an example. For pedagogical purposes, let's assume for this example that our MEMM is only conditioning on the previous tag t_{i-1} and observed word w_i. Concretely, this involves filling an $N \times T$ array with the appropriate values for $P(t_i|t_{i-1}, w_i)$, maintaining backpointers as we proceed. As with HMM Viterbi, when the table is filled, we simply follow pointers back from the maximum value in the final column to retrieve the desired set of labels. The requisite changes from the HMM-style application of Viterbi have to do only with how we fill each cell. Recall from Eq. 9.22 that the recursive step of the Viterbi equation computes the Viterbi value of time t for state j as

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \leq j \leq N, 1 < t \leq T \tag{10.30}$$

which is the HMM implementation of

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) P(s_j|s_i) P(o_t|s_j) \quad 1 \leq j \leq N, 1 < t \leq T \tag{10.31}$$

The MEMM requires only a slight change to this latter formula, replacing the *a* and *b* prior and likelihood probabilities with the direct posterior:

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i) P(s_j|s_i, o_t) \quad 1 \leq j \leq N, 1 < t \leq T \tag{10.32}$$

Figure 10.14 shows an example of the Viterbi trellis for an MEMM applied to the ice-cream task from Section 9.4. Recall that the task is figuring out the hidden weather (hot or cold) from observed numbers of ice creams eaten in Jason Eisner's diary. Figure 10.14 shows the abstract Viterbi probability calculation, assuming that we have a MaxEnt model that computes $P(s_i|s_{i-1}, o_i)$ for us.

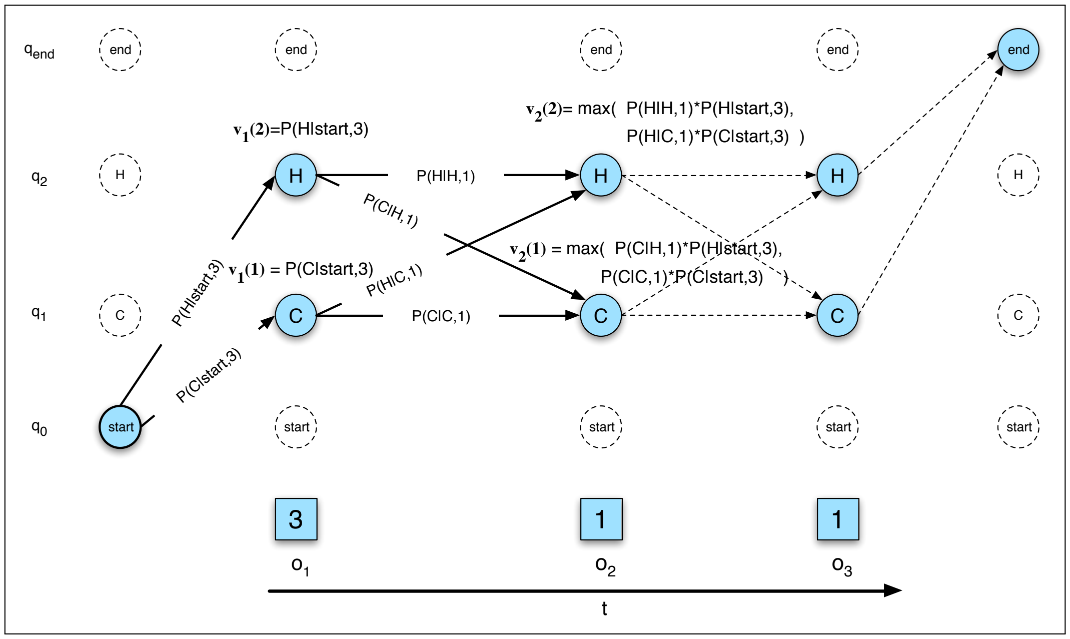

Figure 10.14 Inference from ice-cream eating computed by an MEMM instead of an HMM. The Viterbi trellis for computing the best path through the hidden state space for the ice-cream eating events *3 1 3*, modified from the HMM figure in Fig. 9.10.

Learning in MEMMs relies on the same supervised learning algorithms we presented for logistic regression. Given a sequence of observations, feature functions, and corresponding hidden states, we train the weights so as maximize the log-likelihood of the training corpus. As with logistic regression, regularization is important, and all modern systems use L1 or L2 regularization.

10.6 Bidirectionality

The one problem with the MEMM and HMM models as presented is that they are exclusively run left-to-right. While the Viterbi algorithm still allows present decisions to be influenced indirectly by future decisions, it would help even more if a decision about word w_i could directly use information about future tags t_{i+1} and t_{i+2}.

label bias
observation bias

Adding bidirectionality has another useful advantage. MEMMs have a theoretical weakness, referred to alternatively as the **label bias** or **observation bias** problem (Lafferty et al. 2001, Toutanova et al. 2003). These are names for situations when one source of information is ignored because it is **explained away** by another source. Consider an example from (Toutanova et al., 2003), the sequence *will/NN to/TO fight/VB*. The tag TO is often preceded by NN but rarely by modals (MD), and so that tendency should help predict the correct NN tag for *will*. But the previous transition $P(t_{will}|\langle s \rangle)$ prefers the modal, and because $P(TO|to,t_{will})$ is so close to 1 regardless of t_{will} the model cannot make use of the transition probability and incorrectly chooses MD. The strong information that *to* must have the tag TO has **explained away** the presence of TO and so the model doesn't learn the importance of the previous NN tag for predicting TO. Bidirectionality helps the model by making the link between TO available when tagging the NN.

CRF

One way to implement bidirectionality is to switch to a much more powerful model called a **Conditional Random Field** or **CRF**, which we will introduce in Chapter 20. But CRFs are much more expensive computationally than MEMMs and don't work any better for tagging, and so are not generally used for this task.

Stanford tagger

Instead, other ways are generally used to add bidirectionality. The **Stanford tagger** uses a bidirectional version of the MEMM called a cyclic dependency network (Toutanova et al., 2003).

SVMTool

Alternatively, any sequence model can be turned into a bidirectional model by using multiple passes. For example, the first pass would use only part-of-speech features from already-disambiguated words on the left. In the second pass, tags for all words, including those on the right, can be used. Alternately, the tagger can be run twice, once left-to-right and once right-to-left. In greedy decoding, for each word the classifier chooses the highest-scoring of the tag assigned by the left-to-right and right-to-left classifier. In Viterbi decdoing, the classifier chooses the higher scoring of the two sequences (left-to-right or right-to-left). Multiple-pass decoding is available in publicly available toolkits like the **SVMTool** system (Giménez and Marquez, 2004), a tagger that applies an SVM classifier instead of a MaxEnt classifier at each position, but similarly using Viterbi (or greedy) decoding to implement a sequence model.

10.7 Part-of-Speech Tagging for Other Languages

The HMM and MEMM speech tagging algorithms have been applied to tagging in many languages besides English. For languages similar to English, the methods work well as is; tagger accuracies for German, for example, are close to those for English. Augmentations become necessary when dealing with highly inflected or agglutinative languages with rich morphology like Czech, Hungarian and Turkish.

These productive word-formation processes result in a large vocabulary for these languages: a 250,000 word token corpus of Hungarian has more than twice as many word types as a similarly sized corpus of English (Oravecz and Dienes, 2002), while a 10 million word token corpus of Turkish contains four times as many word types as a similarly sized English corpus (Hakkani-Tür et al., 2002). Large vocabularies mean many unknown words, and these unknown words cause significant performance degradations in a wide variety of languages (including Czech, Slovene, Estonian, and Romanian) (Hajič, 2000).

Highly inflectional languages also have much more information than English coded in word morphology, like **case** (nominative, accusative, genitive) or **gender** (masculine, feminine). Because this information is important for tasks like parsing and coreference resolution, part-of-speech taggers for morphologically rich languages need to label words with case and gender information. Tagsets for morphologically rich languages are therefore sequences of morphological tags rather than a single primitive tag. Here's a Turkish example, in which the word *izin* has three possible morphological/part-of-speech tags and meanings (Hakkani-Tür et al., 2002):

1. Yerdeki **izin** temizlenmesi gerek. iz + `Noun+A3sg+Pnon+Gen`
   The trace on the floor should be cleaned.
2. Üzerinde parmak **izin** kalmiş iz + `Noun+A3sg+P2sg+Nom`
   Your finger **print** is left on (it).
3. Içeri girmek için **izin** alman gerekiyor. izin + `Noun+A3sg+Pnon+Nom`
   You need a **permission** to enter.

Using a morphological parse sequence like `Noun+A3sg+Pnon+Gen` as the part-of-speech tag greatly increases the number of parts-of-speech, and so tagsets can be 4 to 10 times larger than the 50–100 tags we have seen for English. With such large tagsets, each word needs to be morphologically analyzed (using a method from Chapter 3, or an extensive dictionary) to generate the list of possible morphological tag sequences (part-of-speech tags) for the word. The role of the tagger is then to disambiguate among these tags. This method also helps with unknown words since morphological parsers can accept unknown stems and still segment the affixes properly.

Different problems occur with languages like Chinese in which words are not segmented in the writing system. For Chinese part-of-speech tagging word segmentation (Chapter 2) is therefore generally applied before tagging. It is also possible to build sequence models that do joint segmentation and tagging. Although Chinese words are on average very short (around 2.4 characters per unknown word compared with 7.7 for English) the problem of unknown words is still large, although while English unknown words tend to be proper nouns in Chinese the majority of unknown words are common nouns and verbs because of extensive compounding. Tagging models for Chinese use similar unknown word features to English, including character prefix and suffix features, as well as novel features like the **radicals** of each character in a word. One standard unknown feature for Chinese is to build a dictionary in which each character is listed with a vector of each part-of-speech tags that it occurred with in any word in the training set. The vectors of each of the characters in a word are then used as a feature in classification (Tseng et al., 2005b).

10.8 Summary

This chapter introduced the idea of **parts-of-speech** and **part-of-speech tagging**. The main ideas:

- Languages generally have a relatively small set of **closed class** words that are often highly frequent, generally act as **function words**, and can be ambiguous in their part-of-speech tags. Open-class words generally include various kinds of **nouns**, **verbs**, **adjectives**. There are a number of part-of-speech coding schemes, based on **tagsets** of between 40 and 200 tags.
- **Part-of-speech tagging** is the process of assigning a part-of-speech label to each of a sequence of words.
- Two common approaches to **sequence modeling** are a **generative** approach, **HMM** tagging, and a **discriminative** approach, **MEMM** tagging.
- The probabilities in HMM taggers are estimated, not using EM, but directly by maximum likelihood estimation on hand-labeled training corpora. The Viterbi algorithm is used to find the most likely tag sequence
- **Maximum entropy Markov model** or **MEMM taggers** train logistic regression models to pick the best tag given an observation word and its context and the previous tags, and then use Viterbi to choose the best sequence of tags for the sentence. More complex augmentions of the MEMM exist, like the Conditional Random Field (CRF) tagger.
- Modern taggers are generally run **bidirectionally**.

Bibliographical and Historical Notes

What is probably the earliest part-of-speech tagger was part of the parser in Zellig Harris's Transformations and Discourse Analysis Project (TDAP), implemented between June 1958 and July 1959 at the University of Pennsylvania (Harris, 1962), although earlier systems had used part-of-speech information in dictionaries. TDAP used 14 hand-written rules for part-of-speech disambiguation; the use of part-of-speech tag sequences and the relative frequency of tags for a word prefigures all modern algorithms. The parser, whose implementation essentially corresponded a cascade of finite-state transducers, was reimplemented (Joshi and Hopely 1999; Karttunen 1999).

The Computational Grammar Coder (CGC) of Klein and Simmons (1963) had three components: a lexicon, a morphological analyzer, and a context disambiguator. The small 1500-word lexicon listed only function words and other irregular words. The morphological analyzer used inflectional and derivational suffixes to assign part-of-speech classes. These were run over words to produce candidate parts-of-speech which were then disambiguated by a set of 500 context rules by relying on surrounding islands of unambiguous words. For example, one rule said that between an ARTICLE and a VERB, the only allowable sequences were ADJ-NOUN, NOUN-ADVERB, or NOUN-NOUN. The CGC algorithm reported 90% accuracy on applying a 30-tag tagset to a corpus of articles.

The TAGGIT tagger (Greene and Rubin, 1971) was based on the Klein and Simmons (1963) system, using the same architecture but increasing the size of the dictionary and the size of the tagset to 87 tags. TAGGIT was applied to the Brown corpus and, according to Francis and Kučera (1982, p. 9), accurately tagged 77% of the corpus; the remainder of the Brown corpus was then tagged by hand.

All these early algorithms were based on a two-stage architecture in which a dictionary was first used to assign each word a list of potential parts-of-speech and in the second stage large lists of hand-written disambiguation rules winnow down this list to a single part of speech for each word.

Soon afterwards the alternative probabilistic architectures began to be developed. Probabilities were used in tagging by Stolz et al. (1965) and a complete probabilistic tagger with Viterbi decoding was sketched by Bahl and Mercer (1976). The Lancaster-Oslo/Bergen (LOB) corpus, a British English equivalent of the Brown corpus, was tagging in the early 1980's with the CLAWS tagger (Marshall 1983; Marshall 1987; Garside 1987), a probabilistic algorithm that can be viewed as a simplified approximation to the HMM tagging approach. The algorithm used tag bigram probabilities, but instead of storing the word likelihood of each tag, the algorithm marked tags either as *rare* ($P(\text{tag}|\text{word}) < .01$) *infrequent* ($P(\text{tag}|\text{word}) < .10$) or *normally frequent* ($P(\text{tag}|\text{word}) > .10$).

DeRose (1988) developed an algorithm that was almost the HMM approach, including the use of dynamic programming, although computing a slightly different probability: $P(t|w)P(w)$ instead of $P(w|t)P(w)$. The same year, the probabilistic PARTS tagger of Church (1988), (1989) was probably the first implemented HMM tagger, described correctly in Church (1989), although Church (1988) also described the computation incorrectly as $P(t|w)P(w)$ instead of $P(w|t)P(w)$. Church (p.c.) explained that he had simplified for pedagogical purposes because using the probability $P(t|w)$ made the idea seem more understandable as "storing a lexicon in an almost standard form".

Later taggers explicitly introduced the use of the hidden Markov model (Kupiec 1992; Weischedel et al. 1993; Schütze and Singer 1994). Merialdo (1994) showed that fully unsupervised EM didn't work well for the tagging task and that reliance on hand-labeled data was important. Charniak et al. (1993) showed the importance of the most frequent tag baseline; the 92.3% number we give above was from Abney et al. (1999). See Brants (2000) for many implementation details of a state-of-the-art HMM tagger.

Ratnaparkhi (1996) introduced the MEMM tagger, called MXPOST, and the modern formulation is very much based on his work.

The idea of using letter suffixes for unknown words is quite old; the early Klein and Simmons (1963) system checked all final letter suffixes of lengths 1-5. The probabilistic formulation we described for HMMs comes from Samuelsson (1993). The unknown word features described on page 159 come mainly from (Ratnaparkhi, 1996), with augmentations from Toutanova et al. (2003) and Manning (2011).

State of the art taggers are based on a number of models developed just after the turn of the last century, including (Collins, 2002) which used the the perceptron algorithm, Toutanova et al. (2003) using a bidirectional log-linear model, and (Giménez and Marquez, 2004) using SVMs. HMM (Brants 2000; Thede and Harper 1999) and MEMM tagger accuracies are likely just a tad lower.

An alternative modern formalism, the English Constraint Grammar systems (Karlsson et al. 1995; Voutilainen 1995; Voutilainen 1999), uses a two-stage formalism much like the very early taggers from the 1950s and 1960s. A very large morphological analyzer with tens of thousands of English word stems entries is used to return all possible parts-of-speech for a word, using a rich feature-based set of tags. So the word *occurred* is tagged with the options ⟨ V PCP2 SV ⟩ and ⟨ V PAST VFIN SV ⟩, meaning it can be a participle (PCP2) for an intransitive (SV) verb, or a past (PAST) finite (VFIN) form of an intransitive (SV) verb. A large set of 3,744 constraints are then applied to the input sentence to rule out parts-of-speech that are inconsistent with the context. For example here's one rule for the ambiguous word *that*, that eliminates all tags except the ADV (adverbial intensifier) sense (this is the sense in the sentence *it isn't <u>that</u> odd*):

```
ADVERBIAL-THAT RULE
Given input: "that"
if
  (+1 A/ADV/QUANT); /* if next word is adj, adverb, or quantifier */
  (+2 SENT-LIM);      /* and following which is a sentence boundary, */
  (NOT -1 SVOC/A); /* and the previous word is not a verb like */
                          /* 'consider' which allows adjs as object complements */
then eliminate non-ADV tags
else eliminate ADV tag
```

The combination of the extensive morphological analyzer and carefully written constraints leads to a very high accuracy for the constraint grammar algorithm (Samuelsson and Voutilainen, 1997).

Manning (2011) investigates the remaining 2.7% of errors in a state-of-the-art tagger, the bidirectional MEMM-style model described above (Toutanova et al., 2003). He suggests that a third or half of these remaining errors are due to errors or inconsistencies in the training data, a third might be solvable with richer linguistic models, and for the remainder the task is underspecified or unclear.

The algorithms presented in the chapter rely heavily on in-domain training data hand-labeled by experts. Much recent work in part-of-speech tagging focuses on ways to relax this assumption. Unsupervised algorithms for part-of-speech tagging cluster words into part-of-speech-like classes (Schütze 1995; Clark 2000; Goldwater and Griffiths 2007; Berg-Kirkpatrick et al. 2010; Sirts et al. 2014) ; see Christodoulopoulos et al. (2010) for a summary. Many algorithms focus on combining labeled and unlabeled data, for example by co-training (Clark et al. 2003; Søgaard 2010). Assigning tags to text from very different genres like **Twitter** text can involve adding new tags for URLS (URL), username mentions (USR), retweets (RT), and hashtags (HT), normalization of non-standard words, and bootstrapping to employ unsupervised data (Derczynski et al., 2013).

Twitter

Readers interested in the history of parts-of-speech should consult a history of linguistics such as Robins (1967) or Koerner and Asher (1995), particularly the article by Householder (1995) in the latter. Sampson (1987) and Garside et al. (1997) give a detailed summary of the provenance and makeup of the Brown and other tagsets.

Exercises

10.1 Find one tagging error in each of the following sentences that are tagged with the Penn Treebank tagset:

1. I/PRP need/VBP a/DT flight/NN from/IN Atlanta/NN
2. Does/VBZ this/DT flight/NN serve/VB dinner/NNS
3. I/PRP have/VB a/DT friend/NN living/VBG in/IN Denver/NNP
4. Can/VBP you/PRP list/VB the/DT nonstop/JJ afternoon/NN flights/NNS

10.2 Use the Penn Treebank tagset to tag each word in the following sentences from Damon Runyon's short stories. You may ignore punctuation. Some of these are quite difficult; do your best.

1. It is a nice night.
2. This crap game is over a garage in Fifty-second Street...
3. ...Nobody ever takes the newspapers she sells ...

4. He is a tall, skinny guy with a long, sad, mean-looking kisser, and a mournful voice.
5. . . . I am sitting in Mindy's restaurant putting on the gefillte fish, which is a dish I am very fond of, . . .
6. When a guy and a doll get to taking peeks back and forth at each other, why there you are indeed.

10.3 Now compare your tags from the previous exercise with one or two friend's answers. On which words did you disagree the most? Why?

10.4 Implement the "most likely tag" baseline. Find a POS-tagged training set, and use it to compute for each word the tag that maximizes $p(t|w)$. You will need to implement a simple tokenizer to deal with sentence boundaries. Start by assuming that all unknown words are NN and compute your error rate on known and unknown words. Now write at least five rules to do a better job of tagging unknown words, and show the difference in error rates.

10.5 Build a bigram HMM tagger. You will need a part-of-speech-tagged corpus. First split the corpus into a training set and test set. From the labeled training set, train the transition and observation probabilities of the HMM tagger directly on the hand-tagged data. Then implement the Viterbi algorithm from this chapter and Chapter 9 so that you can label an arbitrary test sentence. Now run your algorithm on the test set. Report its error rate and compare its performance to the most frequent tag baseline.

10.6 Do an error analysis of your tagger. Build a confusion matrix and investigate the most frequent errors. Propose some features for improving the performance of your tagger on these errors.

CHAPTER

11 Formal Grammars of English

The study of grammar has an ancient pedigree; Panini's grammar of Sanskrit was written over two thousand years ago and is still referenced today in teaching Sanskrit. Despite this history, knowledge of grammar and syntax remains spotty at best. In this chapter, we make a preliminary stab at addressing some of these gaps in our knowledge of grammar and syntax, as well as introducing some of the formal mechanisms that are available for capturing this knowledge in a computationally useful manner.

Syntax

The word **syntax** comes from the Greek *sýntaxis*, meaning "setting out together or arrangement", and refers to the way words are arranged together. We have seen various syntactic notions in previous chapters. The regular languages introduced in Chapter 2 offered a simple way to represent the ordering of strings of words, and Chapter 4 showed how to compute probabilities for these word sequences. Chapter 10 showed that part-of-speech categories could act as a kind of equivalence class for words. This chapter and the ones that follow introduce a variety of syntactic phenomena as well as form models of syntax and grammar that go well beyond these simpler approaches.

The bulk of this chapter is devoted to the topic of context-free grammars. Context-free grammars are the backbone of many formal models of the syntax of natural language (and, for that matter, of computer languages). As such, they are integral to many computational applications, including grammar checking, semantic interpretation, dialogue understanding, and machine translation. They are powerful enough to express sophisticated relations among the words in a sentence, yet computationally tractable enough that efficient algorithms exist for parsing sentences with them (as we show in Chapter 12). In Chapter 13, we show that adding probability to context-free grammars gives us a powerful model of disambiguation. And in Chapter 20 we show how they provide a systematic framework for semantic interpretation.

In addition to an introduction to this grammar formalism, this chapter also provides a brief overview of the grammar of English. To illustrate our grammars, we have chosen a domain that has relatively simple sentences, the Air Traffic Information System (ATIS) domain (Hemphill et al., 1990). ATIS systems were an early example of spoken language systems for helping book airline reservations. Users try to book flights by conversing with the system, specifying constraints like *I'd like to fly from Atlanta to Denver*.

11.1 Constituency

The fundamental notion underlying the idea of constituency is that of abstraction — groups of words behaving as a single units, or constituents. A significant part of developing a grammar involves discovering the inventory of constituents present in the language.