# 6 HIDDEN MARKOV AND MAXIMUM ENTROPY MODELS

*Numquam ponenda est pluralitas sine necessitat*
'Plurality should never be proposed unless needed'
William of Occam

*Tatyana was her name... I own it,*
*self-willed it may be just the same;*
*but it's the first time you'll have known it,*
*a novel graced with such a name*
Pushkin, *Eugene Onegin*

In this chapter we introduce two important classes of statistical models for processing text and speech, the **Hidden Markov Model** (**HMM**) and the **Maximum Entropy** model (**MaxEnt**), particularly a variant of MaxEnt called the **Maximum Entropy Markov Model** (**MEMM**). All of these are **machine learning** models. We have already touched on some aspects of machine learning; indeed we briefly introduced the Hidden Markov Model in the previous chapter, and we have introduced the *N*-gram model in the chapter before. In this chapter we give a more complete and formal introduction to these two important models.

SEQUENCE CLASSIFIERS

HMMs and MEMMs are both **sequence classifiers**. A sequence classifier or **sequence labeler** is a model whose job is to assign some label or class to each unit in a sequence. The finite-state transducer we studied in Ch. 3 is a kind of non-probabilistic sequence classifier, for example transducing from sequences of words to sequences of morphemes. The HMM and MEMM extend this notion by being probabilistic sequence classifiers; given a sequence of units (words, letters, morphemes, sentences, whatever) their job is to compute a probability distribution over possible labels and choose the best label sequence.

We have already seen one important sequence classification task: part-of-speech tagging, where each word in a sequence has to be assigned a part-of-speech tag. 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. Besides part-of-speech tagging, in this book we will see the application of these sequence models to tasks like speech recognition (Ch. 9), sentence segmentation and grapheme-to-phoneme conversion (Ch. 8), partial parsing/chunking (Ch. 12), and named entity recognition and information extraction (Ch. 17).

This chapter is roughly divided into two sections: Hidden Markov Models followed by Maximum Entropy Markov Models. Our discussion of the Hidden Markov Model extends what we said about HMM part-of-speech tagging. We begin in the next section by introducing the Markov Chain, then give a detailed overview of HMMs and the forward and Viterbi algorithms with more formalization, and finally introduce the important EM algorithm for unsupervised (or semi-supervised) learning of a Hidden Markov model.

In the second half of the chapter, we introduce Maximum Entropy Markov Models gradually, beginning with techniques that may already be familiar to you from statistics: linear regression and logistic regression. We next introduce MaxEnt. MaxEnt by itself is not a sequence classifier; it is used to assign a class to a single element. The name Maximum Entropy comes from the idea that the classifier finds the probabilistic model which follows Occam's Razor in being the simplest (least constrained; has the maximum entropy) yet still consistent with some specific constraints. The Maximum Entropy Markov Model is the extension of MaxEnt to the sequence labeling task, adding components such as the Viterbi algorithm.

Although this chapter introduces MaxEnt, which is a classifier, we will not focus in general on non-sequential classification. Non-sequential classification will be addressed in later chapters with the introduction of classifiers like the **Gaussian Mixture Model** in (Ch. 9) and the **Naive Bayes** and **decision list** classifiers in (Ch. 19).

## 6.1 Markov Chains

The Hidden Markov Model is one of the most important machine learning models in speech and language processing. In order 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 Ch. 3. Recall that a finite automaton is defined by a set of states, and a set of transitions between states that are taken based on the input observations. A **weighted finite-state automaton** is a simple augmentation of the finite automaton in which each arc is associated with a probability, indicating how likely that path is to be taken. The probability on all the arcs leaving a node must sum to 1.

WEIGHTED

MARKOV CHAIN

A **Markov chain** is a special case of a weighted automaton in which the input sequence uniquely determines which states the automaton will go through. Because they can't represent inherently ambiguous problems, a Markov chain is only useful for assigning probabilities to unambiguous sequences.

Fig. 6.1a shows a Markov chain for assigning a probability to a sequence of weather events, where the vocabulary consists of HOT, COLD, and RAINY,. Fig. 6.1b 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

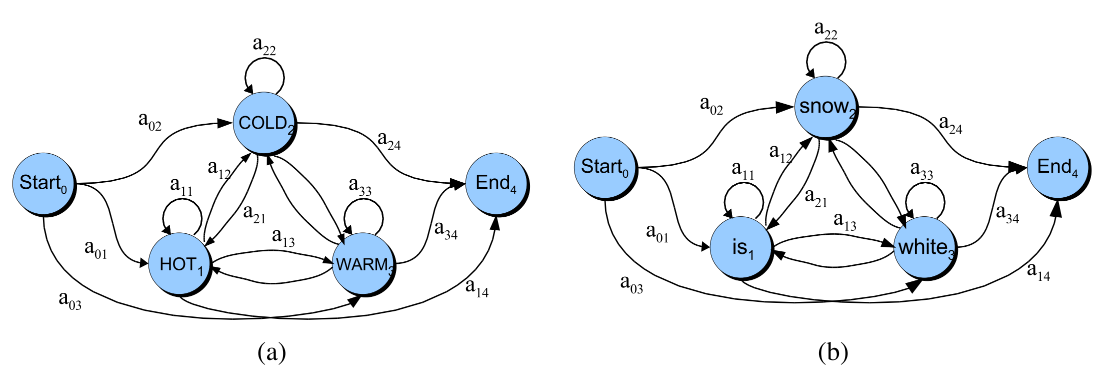

**Figure 6.1** A Markov chain for weather (a) and one for words (b). A Markov chain is specified by the structure, the transition between states, and the start and end states.

bigram language model. Given the two models in Figure 6.1 we can assign a probability to any sequence from our vocabulary. We'll go over how to do this shortly.

First, let's be more formal. We'll 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 **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. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$ |
| $q_0, q_{end}$ | a special **start and end state** which are not associated with observations. |

Fig. 6.1 shows that we represent the states (including start and end states) as nodes in the graph, and the transitions as edges between nodes.

FIRST-ORDER

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

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

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:

(6.2) $$\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$$

An alternate 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. $\pi_i$ 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 $\pi_1$. Note that because each $\pi_i$ expresses the probability $p(q_i|START)$, all the $\pi$ probabilities must sum to 1:

$$\sum_{i=1}^{n} \pi_i = 1 \qquad (6.3)$$

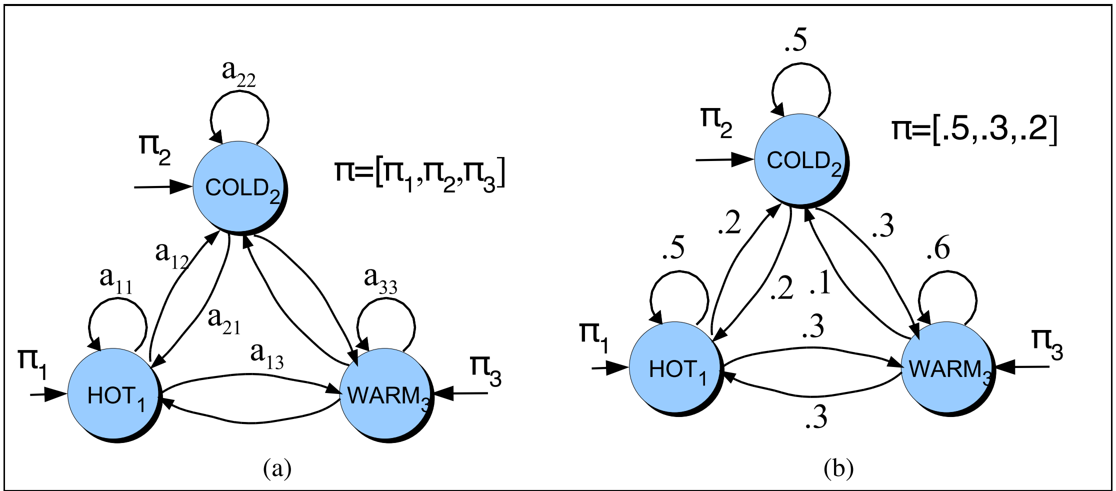

**Figure 6.2** Another representation of the same Markov chain for weather shown in Fig. 6.1. Instead of using a special start state with $a_{01}$ transition probabilities, we use the $\pi$ vector, which represents the distribution over starting state probabilities. The figure in (b) shows sample probabilities.

Before you go on, use the sample probabilities in Fig. 6.2b to compute the probability of each of the following sequences:

(6.4) hot hot hot hot

(6.5) cold hot cold hot

What does the difference in these probabilities tell you about a real-world weather fact encoded in Fig. 6.2b?

## 6.2 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 for part-of-speech tagging (Ch. 5) 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. We will see the same thing in speech recognition; we'll 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 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.

HIDDEN MARKOV 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 by seeing how a Hidden Markov Model differs from a Markov chain. An **HMM** is specified by a set of **states** $Q$, a set of **transition probabilities** $A$, a set of observation likelihoods $B$, a defined **start state** and **end state(s)**, and a set of **observation symbols** $O$, which is not drawn from the same alphabet as the state set $Q$:

HMM

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:

HMM

| | |
|---|---|
| $Q = q_1 q_2 \ldots q_N$ | a set of **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. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$ |
| $O = o_1 o_2 \ldots o_N$ | a set of **observations**, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$. |
| $B = b_i(o_t)$ | A set of **observation likelihoods:**, also called **emission probabilities**, each expressing the probability of an observation $o_t$ being generated from a state $i$. |
| $q_0, q_{end}$ | a special **start and end state** which are not associated with observation. |

As we noted for Markov chains, an alternate 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:

$\pi = \pi_1, \pi_2, ..., \pi_N$ an **initial probability distribution** over states. $\pi_i$ 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 is dependent only on the previous state:

(6.6) **Markov Assumption:** $P(q_i|q_1...q_{i-1}) = P(q_i|q_{i-1})$

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

**Output Independence Assumption:** $P(o_i|q_1 \dots q_i, \dots, q_n, o_1, \dots, o_i, \dots, o_n) = P(o_i|q_i)$

(6.7)

Fig. 6.3 shows a sample HMM for the ice cream task. The two hidden states (H and C) correspond to hot and cold weather, while the observations (drawn from the alphabet $O = \{1,2,3\}$) correspond to the number of ice creams eaten by Jason on a given day.

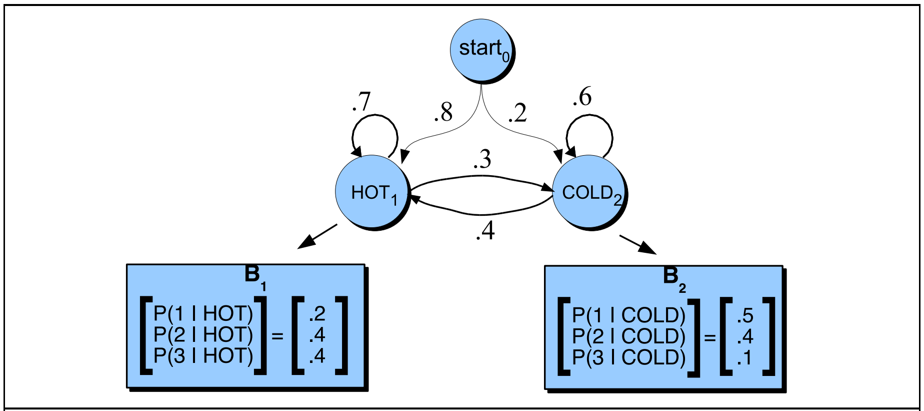
start0
.7
.8
.2
.6
.3
HOT1
COLD2
.4
B1
P(1 I HOT) P(2 I HOT) P(3 I HOT) = .2 .4 .4
B2
P(1 I COLD) P(2 I COLD) P(3 I COLD) = .5 .4 .1

**Figure 6.3** A Hidden Markov Model for relating numbers of ice creams eaten by Jason (the observations) to the weather (H or C, the hidden variables). Note that we have used a special zeroth start state; we could instead have represented the initial distribution over states by using the $\pi$ vector, $\pi = [.8, .2]$. In general we will use the start state rather than the $\pi$ vector in the remainder of this chapter.

FULLY-CONNECTED
ERGODIC HMM
LEFT-TO-RIGHT
BAKIS

Notice that in the HMM in Fig. 6.3, 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. 6.4. In a Bakis HMM, there are no transitions going 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 will see more of this in Ch. 9.

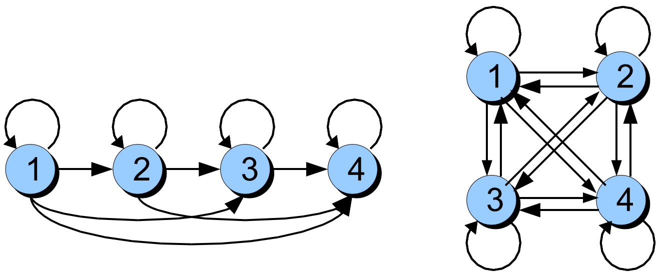

**Figure 6.4** Two 4-state Hidden Markov models; a left-to-right (Bakis) HMM on the left, and a fully-connected (ergodic) HMM on the right. In the Bakis model, all transitions not shown have zero probability.

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 (Computing 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 Ch. 5; now in the next three sections we introduce all three tasks more formally.

## 6.3 COMPUTING LIKELIHOOD: THE FORWARD ALGORITHM

Our first problem is to compute the likelihood of a particular observation sequence. For example, given the HMM in Fig. 6.2b, 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. [1]

Given this one-to-one mapping, and the Markov assumptions expressed in Eq. 6.6, for a particular hidden state sequence $Q = q_0, q_1, q_2, ..., q_n$ and an observation sequence $O = o_1, o_2, ..., o_n$, the likelihood of the observation sequence (using a special start state $q_0$ rather than $\pi$ probabilities) is:

$$P(O|Q) = \prod_{i=1}^{n} P(o_i|q_i) \times \prod_{i=1}^{n} P(q_i|q_{i-1}) \tag{6.8}$$

The computation of the forward probability for our ice-cream observation *3 1 3* from one possible hidden state sequence *hot hot hot* is as follows (Fig. 6.5 shows a graphic representation of this):

$$\begin{aligned} P(3\ 1\ 3|\text{hot hot cold}) \ = \ & P(\text{hot}|\text{start}) \times P(\text{hot}|\text{hot}) \times P(\text{cold}|\text{hot}) \\ & \times P(3|\text{hot}) \times P(1|\text{hot}) \times P(3|\text{cold}) \end{aligned} \tag{6.9}$$

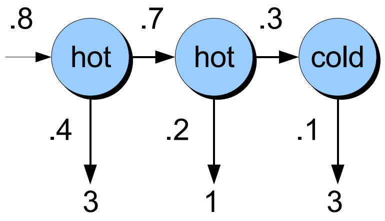

**Figure 6.5** The computation of the observation likelihood for the ice-cream events *3 1 3* given the hidden state sequence *hot hot cold*.

In order to compute the true total likelihood of *3 1 3*, however, we need to sum over all possible hidden state sequences (in this case, the 8 sequences *cold cold cold*, *cold cold hot*, and so on). 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, and so we cannot compute

[1] There are variants of HMMs called **segmental HMMs** (in speech recognition) or **semi-HMMs** (in natural language processing) in which this one-to-one mapping between the length of the hidden state sequence and the length of the observation sequence does not hold.

the total observation likelihood by computing a separate observation likelihood for each hidden state sequence and then summing them up.

FORWARD ALGORITHM

Instead of using such an extremely exponential algorithm, we use an efficient algorithm called the **forward algorithm**.The forward algorithm is a kind of **dynamic programming** algorithm, i.e., 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**.

Fig. 6.6 shows an example of the forward trellis for computing the likelihood of *3 1 3* given the hidden state sequence *hot hot cold*.

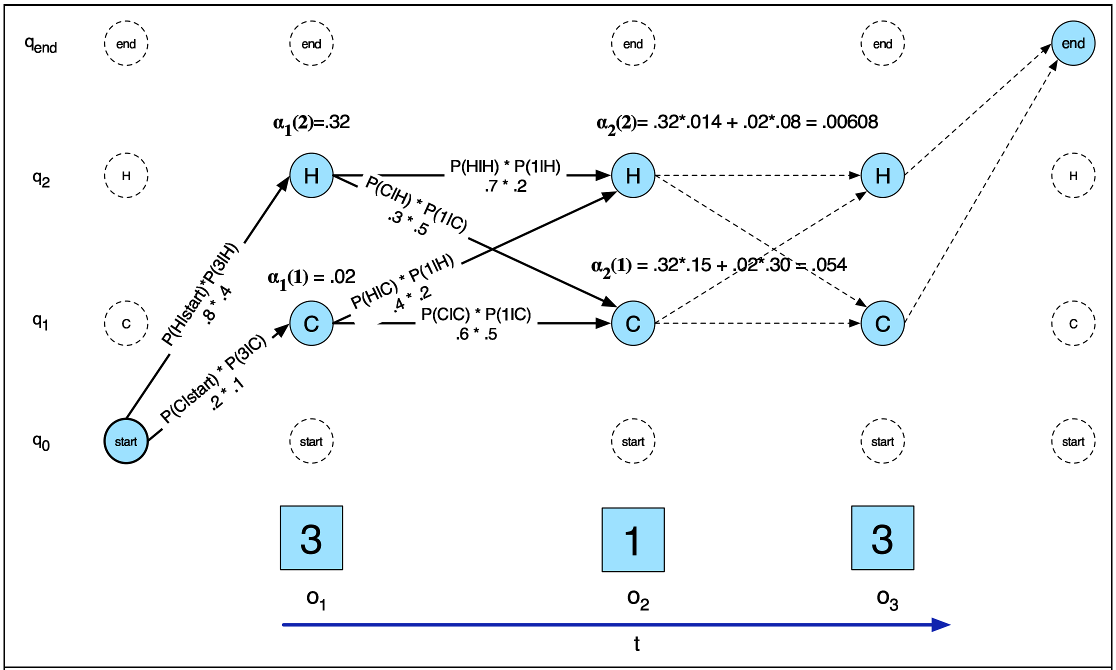

**Figure 6.6** The forward trellis for computing the total observation likelihood for the ice-cream events *3 1 3*. Hidden states are in circles, observations in squares. White (unfilled) circles indicate illegal transitions. The figure shows the computation of $\alpha_t(j)$ for two states at two time steps. The computation in each cell follows Eq˙ 6.10: $\alpha_t(j) = \sum_{i=1}^{N-1} \alpha_{t-1}(i) a_{ij} b_j(o_t)$. The resulting probability expressed in each cell is Eq˙ 6.11: $\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)$.

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 $\lambda$. 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:

$$\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda) \tag{6.10}$$

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

$$\alpha_t(j) = \sum_{i=1}^{N-1} \alpha_{t-1}(i) a_{ij} b_j(o_t) \tag{6.11}$$

The three factors that are multiplied in Eq˙ 6.11 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 $j$

Consider the computation in Fig. 6.6 of $\alpha_2(1)$, the forward probability of being at time step 2 in state 1 having generated the partial observation *3 1*. This is computed by extending the $\alpha$ 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 $\alpha_1(2) \times P(H|C) \times P(1|H)$.

Fig. 6.7 shows another visualization of this induction step for computing the value in one new cell of the trellis.

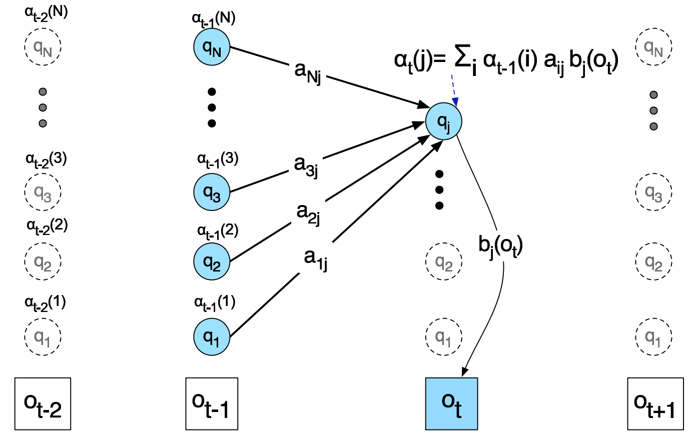

**Figure 6.7** Visualizing the computation of a single element $\alpha_t(i)$ in the trellis by summing all the previous values $\alpha_{t-1}$ weighted by their transition probabilities $a$ and multiplying by the observation probability $b_i(o_{t+1})$. For many applications of HMMs, many of the transition 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. Elements which take place in the computation are in blue.

We give two formal definitions of the forward algorithm; the pseudocode in Fig. 6.8 and a statement of the definitional recursion here:

```
function FORWARD(observations of len $T$, state-graph) returns forward-probability

  num-states ← NUM-OF-STATES(state-graph)
  Create a probability matrix forward[num-states+2,T+2]
  forward[0,0] ← 1.0
  for each time step $t$ from 1 to $T$ do
    for each state $s$ from 1 to num-states do
      forward[s,t] ← $\sum_{1 \leq s' \leq \text{num-states}}$ forward[$s'$, $t-1$] $*$ $a_{s',s}$ $*$ $b_s(o_t)$
  return the sum of the probabilities in the final column of forward
```

**Figure 6.8** The forward algorithm; we've used the notation *forward*$[s,t]$ to represent $\alpha_t(s)$.

1. Initialization:

$$\alpha_1(j) \;=\; a_{0j}b_j(o_1) \;\; 1 \leq j \leq N \tag{6.12}$$

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

$$\alpha_t(j) = \sum_{i=1}^{N-1} \alpha_{t-1}(i)a_{ij}b_j(o_t); \quad 1 < j < N, 1 < t < T \tag{6.13}$$

3. Termination:

$$P(O|\lambda) = \alpha_T(N) = \sum_{i=2}^{N-1} \alpha_T(i)\, a_{iN} \tag{6.14}$$

## 6.4 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 $Q = q_1 q_2 q_3 \ldots q_T$.

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 max 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

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

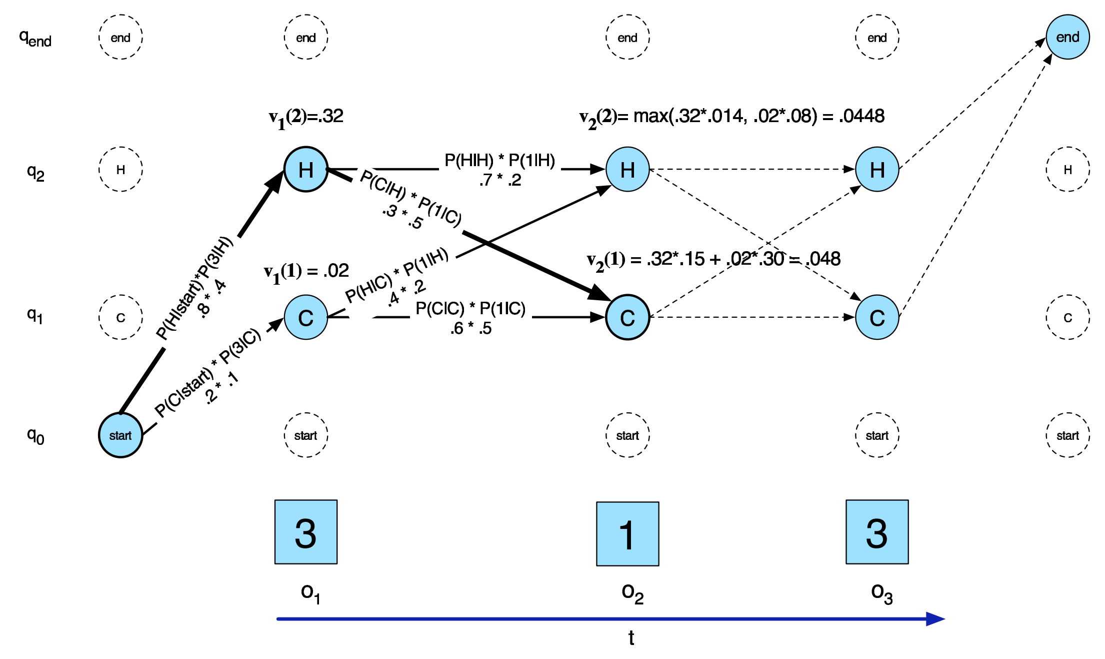

**Figure 6.9** The Viterbi trellis for computing the best path through the hidden state space for the ice-cream eating events *3 1 3*. Hidden states are in circles, observations in squares. White (unfilled) circles indicate illegal transitions. The figure shows the computation of $v_t(j)$ for two states at two time steps. The computation in each cell follows Eq˙ 6.10: $v_t(j) = \max_{1 \leq i \leq N-1} v_{t-1}(i)\, a_{ij}\, b_j(o_t)$ The resulting probability expressed in each cell is Eq˙ 6.16: $v_t(j) = P(q_0, q_1, \ldots, q_{t-1}, o_1, o_2, \ldots, o_t, q_t = j|\lambda)$.

Fig. 6.9 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 Viterbi 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 likely state sequence $q_1 ... q_{t-1}$, given the automaton $\lambda$. 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 following probability:

(6.15)
$$v_t(j) = P(q_0, q_1 ... q_{t-1}, o_1, o_2 \ldots o_t, q_t = j|\lambda)$$

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:

$$v_t(j) \;=\; \max_{1\leq i\leq N-1} v_{t-1}(i)\, a_{ij}\, b_j(o_t) \tag{6.16}$$

The three factors that are multiplied in Eq. 6.16 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 $j$

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

  num-states ← NUM-OF-STATES(state-graph)
  Create a path probability matrix viterbi[num-states+2,T+2]
  viterbi[0,0] ← 1.0
  for each time step t from 1 to T do
    for each state s from 1 to num-states do
      viterbi[s,t] ← $\max_{1 \leq s' \leq num\text{-}states} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
      backpointer[s,t] ← $\operatorname{argmax}_{1 \leq s' \leq num\text{-}states} viterbi[s',t-1] * a_{s',s}$
  Backtrace from highest probability state in final column of viterbi[] and return path
```

**Figure 6.10** 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 which assigns maximum likelihood to the observation sequence. Note that states 0 and N+1 are non-emitting *start* and *end* states.

Fig. 6.10 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 where forward takes the **sum**. Note also that the Viterbi algorithm has one component that the forward algorithm doesn't have: **backpointers**. This is because 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. 6.11.

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

1. **Initialization:**

$$v_1(j) \;=\; a_{0j} b_j(o_1) \;\; 1 \leq j \leq N \tag{6.17}$$

$$bt_1(j) \;=\; 0 \tag{6.18}$$

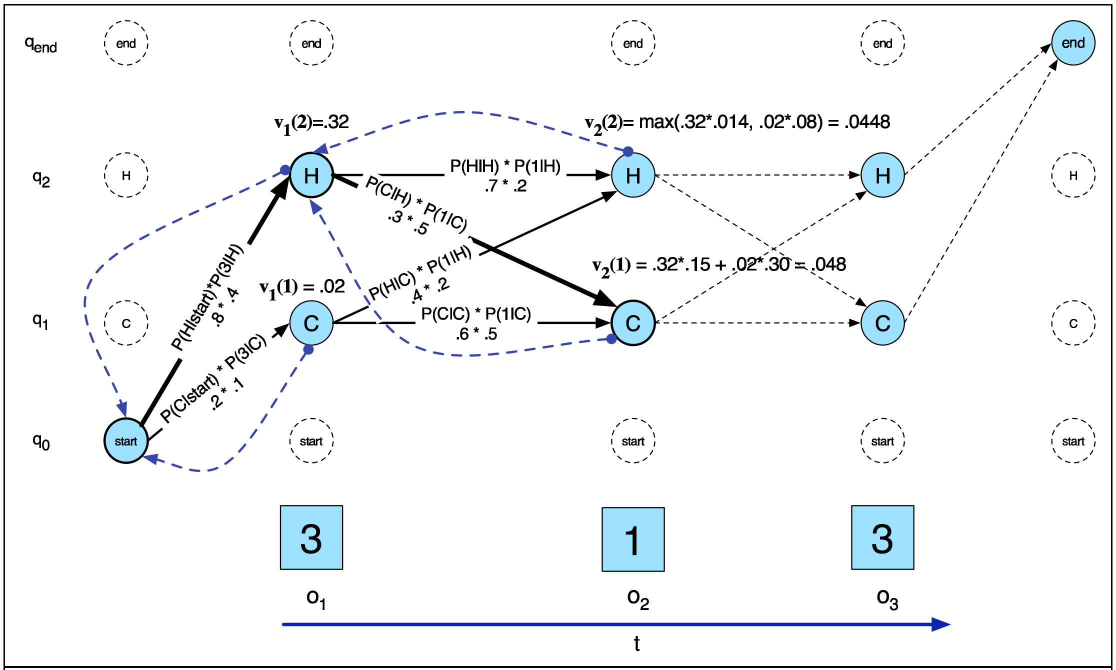

**Figure 6.11** The Viterbi backtrace. As we extend each path to a new state account for the next observation, we keep a backpointer (shown with broken blue lines) to the best path that led us to this state.

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

$$v_t(j) = \max_{1 \le i \le N-1} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 < j < N, 1 < t < T \tag{6.19}$$

$$bt_t(j) = \operatorname*{argmax}_{1 \le i \le N-1} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 < j < N, 1 < t < T \tag{6.20}$$

3. **Termination:**

$$\text{The best score:} \quad P* = \max_{1 \le i \le N} v_T(i) \tag{6.21}$$

$$\text{The start of backtrace:} \quad q_T* = \operatorname*{argmax}_{1 \le i \le N} bt_T(i) \tag{6.22}$$

## 6.5 Training HMMs: The Forward-Backward Algorithm

We turn to the third problem for HMMs: learning the parameters of an HMM, i.e., 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 would start with a sequence of observations $O = \{w_1, w_2, w_3 \ldots\}$ and a set of hidden states *NN, NNS, VBD, IN,...* 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.

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 $C(i \rightarrow j)$, and then normalizing by the total count of all times we took any transition from state $i$:

$$a_{ij} = \frac{C(i \rightarrow j)}{\sum_{q \in Q} C(i \rightarrow q)} \tag{6.23}$$

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

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

The backward probability $\beta$ is the probability of seeing the observations from time $t+1$ to the end, given that we are in state $j$ at time $t$ (and of course given the automaton $\lambda$):

$$\beta_t(i) = P(o_{t+1}, o_{t+2} \ldots o_T | q_t = i, \lambda) \tag{6.24}$$

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

1. **Initialization:**

$$\beta_T(i) \;=\; a_{iN}, \quad 1 < i < N \tag{6.25}$$

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

$$\beta_t(i) = \sum_{i=1}^{N-1} a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j), \quad 1 < i < N, 0 < t < T \tag{6.26}$$

3. **Termination:**

$$P(O|\lambda) = \alpha_T(N) = \beta_T(1) = \sum_{j=1}^{N-1} a_{1j}\, b_j(o_1)\, \beta_1(j) \tag{6.27}$$

Fig. 6.12 illustrates the backward induction step.

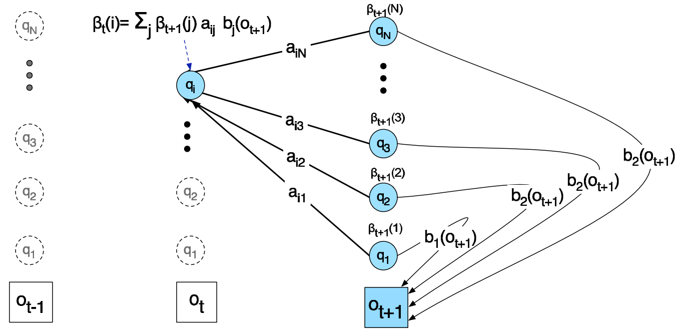

**Figure 6.12** The computation of $\beta_t(i)$ by summing all the successive values $\beta_{t+1}(j)$ weighted by their transition probabilities $a_i j$ and their observation probabilities $b_j(o_{t+1})$.

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 showing how to reestimate $a_{ij}$. We will proceed to estimate $\hat{a}_{ij}$ by a variant of (6.23):

$$\hat{a}_{ij} = \frac{\text{expected number of transitions from state } i \text{ to state } j}{\text{expected number of transitions from state } i} \tag{6.28}$$

How do we compute the numerator? Here's the intuition. Assume we had some estimate of the probability that a given transition $i \rightarrow j$ 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 $i \rightarrow j$.

More formally, let's define the probability $\xi_t$ 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:

$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j|O,\lambda) \tag{6.29}$$

In order to compute $\xi_t$, we first compute a probability which is similar to $\xi_t$, but differs in including the probability of the observation:

$$\text{not-quite-}\xi_t(i,j) = P(q_t = i, q_{t+1} = j, O|\lambda) \tag{6.30}$$

Fig. 6.13 shows the various probabilities that go into computing not-quite-$\xi_t$: the transition probability for the arc in question, the $\alpha$ probability before the arc, the $\beta$ probability after the arc, and the observation probability for the symbol just after the arc.

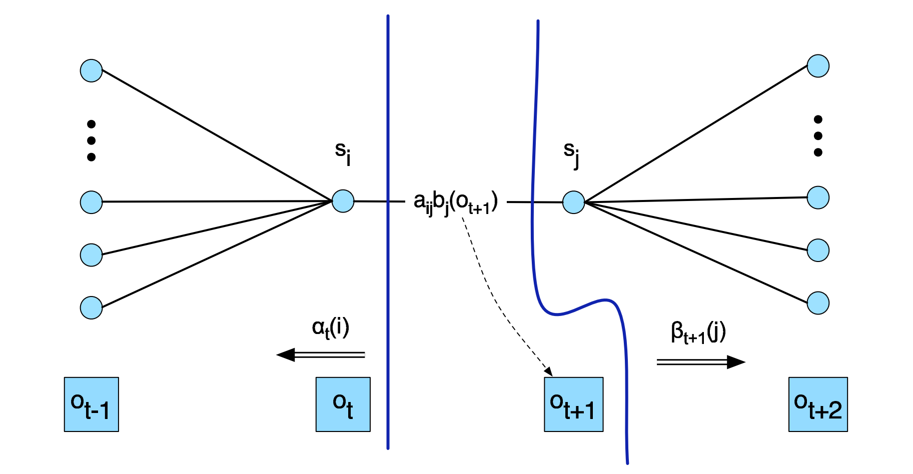

**Figure 6.13** Computation of the joint probability of being in state $i$ at time $t$ and state $j$ at time $t+1$. The figure shows the various probabilities that need to be combined to produce $P(q_t = i, q_{t+1} = j, O|\lambda)$: the $\alpha$ and $\beta$ probabilities, the transition probability $a_{ij}$ and the observation probability $b_j(o_{t+1})$. After Rabiner (1989).

These are multiplied together to produce *not-quite-*$\xi_t$ as follows:

$$\text{not-quite-}\xi_t(i,j) = \alpha_t(i)\,a_{ij}b_j(o_{t+1})\beta_{t+1}(j) \tag{6.31}$$

In order to compute $\xi_t$ from *not-quite-*$\xi_t$, the laws of probability instruct us to divide by $P(O|\lambda)$, since:

$$P(X|Y,Z) = \frac{P(X,Y|Z)}{P(Y|Z)} \tag{6.32}$$

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:

$$P(O|\lambda) = \alpha_T(N) = \beta_T(1) = \sum_{j=1}^{N} \alpha_t(j)\beta_t(j) \tag{6.33}$$

So, the final equation for $\xi_t$ is:

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(N)} \tag{6.34}$$

The expected number of transitions from state $i$ to state $j$ is then the sum over all $t$ of $\xi$. For our estimate of $a_{ij}$ in (6.28), 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 $\hat{a}_{ij}$:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{j=1}^{N}\xi_t(i,j)} \tag{6.35}$$

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:

$$\hat{b}_j(v_k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} \tag{6.36}$$

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

$$\gamma_t(j) = P(q_t = j|O,\lambda) \tag{6.37}$$

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

$$\gamma_t(j) = \frac{P(q_t = j, O|\lambda)}{P(O|\lambda)} \tag{6.38}$$

As Fig. 6.14 shows, the numerator of (6.38) is just the product of the forward probability and the backward probability:

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)} \tag{6.39}$$

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 will be the percentage of the times that we were in state $j$ that we 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 $v_k$):

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t. O_t=v_k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)} \tag{6.40}$$

We now have ways in (6.35) and (6.40) 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 $B$.

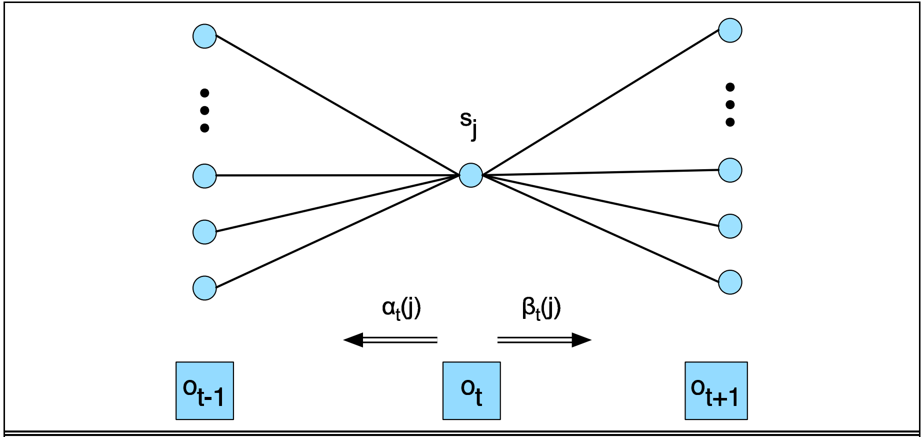

**Figure 6.14** The computation of $\gamma_t(j)$, the probability of being in state $j$ at time $t$. Note that $\xi$ is really a degenerate case of $\xi$ and hence this figure is like a version of Fig. 6.13 with state $i$ collapsed with state $j$. After Rabiner (1989).

These re-estimations form the core of the iterative forward-backward algorithm.

The forward-backward algorithm 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**.

EXPECTATION
E-STEP
MAXIMIZATION
M-STEP

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

Although in principle the forward-backward algorithm can do completely unsupervised learning of the $A$, $B$, and $\pi$ 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 very often set by hand, and only the emission ($B$) and (non-zero) $A$ transition probabilities are trained from a set of observation sequences $O$. Sec. **??** will also discuss how initial estimates for $a$ and $b$ are derived in speech recognition. We will also see in Ch. 9 that the forward-backward algorithm can be extended to inputs which are non-discrete (“continuous observation densities”).

## 6.6 Maximum Entropy Models: Background

We turn now to a second probabilistic machine learning framework called **Maximum Entropy** modeling, **MaxEnt** for short. MaxEnt is more widely known as **multinomial logistic regression**.

Our goal in this chapter is to introduce the use of MaxEnt for sequence classification. Recall that the task of sequence classification or sequence labelling is to assign a label to each element in some sequence, such as assigning a part-of-speech tag

```
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
      $\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)} \ \forall\, t \text{ and } j$
      $\xi_t(i,j) = \frac{\alpha_t(i)\,a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(N)} \ \forall\, t,\ i, \text{ and } j$
    M-step
      $\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}$
      $\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t.\ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$
  return $A, B$
```

**Figure 6.15** The forward-backward algorithm.

to a word. The most common MaxEnt sequence classifier is the **Maximum Entropy Markov Model** or **MEMM**, to be introduced in Sec. 6.8. But before we see this use of MaxEnt as a sequence classifier, we need to introduce non-sequential classification.

The task of classification is to take a single observation, extract some useful features describing the observation, and then based on these features, to **classify** the observation into one of a set of discrete classes. A **probabilistic** classifier does slightly more than this; in addition to assigning a label or class, it gives the **probability** of the observation being in that class; indeed, for a given observation a probabilistic classifier gives a probability distribution over all classes.

Such non-sequential classification tasks occur throughout speech and language processing. For example, in **text classification** we might need to decide whether a particular email should be classified as spam or not. In **sentiment analysis** we have to determine whether a particular sentence or document expresses a positive or negative **opinion**. In many tasks, we'll need to know where the sentence boundaries are, and so we'll need to classify a period character ('.') as either a sentence boundary or not. We'll see more examples of the need for classification throughout this book.

EXPONENTIAL

MaxEnt belongs to the family of classifiers known as the **exponential** or **log-**

LOG-LINEAR

**linear** classifiers. MaxEnt works by extracting some set of features from the input, combining them **linearly** (meaning that we multiply each by a weight and then add them up), and then, for reasons we will see below, using this sum as an exponent.

Let's flesh out this intuition just a bit more. Assume that we have some input $x$ (perhaps it is a word that needs to be tagged, or a document that needs to be classified) from which we extract some features. A feature for tagging might be *this word ends in -ing* or *the previous word was 'the'*. For each such feature $f_i$, we have some weight $w_i$.

Given the features and weights, our goal is to chose a class (for example a part-of-speech tag) for the word. MaxEnt does this by choosing the most probable tag; the probability of a particular class $c$ given the observation $x$ is:

$$p(c|x) = \frac{1}{Z}\exp(\sum_i w_i f_i) \tag{6.41}$$

Here $Z$ is a normalizing factor, used to make the probabilities correctly sum to 1; and as usual $\exp(x) = e^x$.

In order to explain the details of the MaxEnt classifier, including the definition of the normalizing term $Z$ and the intuition of the exponential function, we'll need to understand first **linear regression**, which lays the groundwork for prediction using features, and **logistic regression**, which is our introduction to exponential models. We cover these areas in the next two sections. Readers who have had a grounding in these kinds of regression may want to skip the next two sections. Then in Sec. 6.7 we introduce the details of the MaxEnt classifier. Finally in Sec. 6.8 we show how the MaxEnt classifier is used for sequence classification in the **Maximum Entropy Markov Model** or **MEMM**.

### 6.6.1 Linear Regression

In statistics we use two different names for tasks that map some input features into some output value; we use the word **regression** when the output is real-valued, and **classification** when the output is one of a discrete set of classes.

You may already be familiar with linear regression from a statistics class. The idea is that we are given a set of observations, each observation associated with some features, and we want to predict some real-valued outcome for each observation. Let's see an example from the domain of predicting housing prices. Levitt and Dubner (2005) showed that the words used in a real estate ad can be used as a good predictor of whether a house will sell for more or less than its asking price. They showed, for example, that houses whose real estate ads had words like *fantastic*, *cute*, or *charming*, tended to sell for lower prices, while houses whose ads had words like *maple* and *granite* tended to sell for higher prices. Their hypothesis was that real estate agents used vague positive words like *fantastic* to mask the lack of any specific positive qualities in the house. Just for pedagogical purposes, we created the fake data in Fig. 6.16.

REGRESSION LINE

Fig. 6.17 shows a graph of these points, with the feature (# of adjectives) on the x-axis, and the price on the y-axis. We have also plotted a **regression line**, which is the line that best fits the observed data. The equation of any line is $y = mx + b$; as we show on the graph, the slope of this line is $m = -4900$, while the intercept is 16550.

| Number of vague adjectives | Amount house sold over asking price |
|---|---|
| 4 | 0 |
| 3 | $1000 |
| 2 | $1500 |
| 2 | $6000 |
| 1 | $14000 |
| 0 | $18000 |

**Figure 6.16** Some made-up data on the the number of vague adjectives (*fantastic*, *cute*, *charming*) in a real-estate ad, and the amount the house sold for over the asking price.

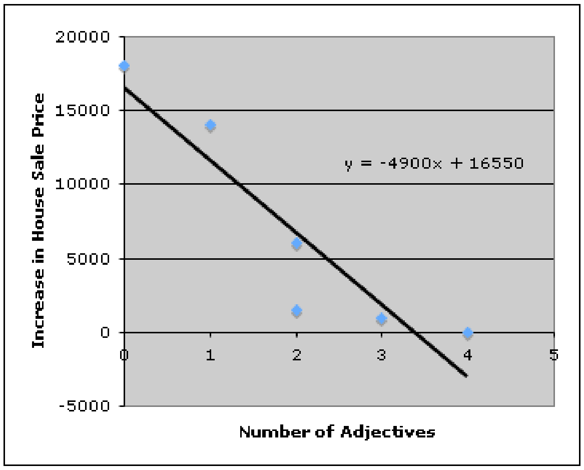

**Figure 6.17** caption

We can think of these two parameters of this line (slope *m* and intercept *b*) as a set of weights that we use to map from our features (in this case *x*, numbers of adjectives) to our output value *y* (in this case price). We can represent this linear function using *w* to refer to weights as follows:

$$\text{price} = w_0 + w_1 * \text{Num\_Adjectives} \tag{6.42}$$

Thus Eq. 6.42 gives us a linear function that lets us estimate the sales price for any number of these adjectives. For example, how much would we expect a house whose ad has 5 adjectives to sell for?

The true power of linear models comes when we use more than one feature (technically we call this **multiple linear regression**). For example, the final house price probably depends on many factors such as the average mortgage rate that month, the number of unsold houses on the market, and many other such factors. We could encode each of these as a variable, and the importance of each factor would be the weight on that variable, as follows:

(6.43) $$\text{price} = w_0 + w_1 * \text{Num\_Adjectives} + w_2 * \text{Mortgage Rate} + w_3 * \text{Num\_Unsold\_Houses}$$

FEATURE

In speech and language processing, we often call each of these predictive factors like the number of adjectives or the mortgage rate a **feature**. We represent each observation (each house for sale) by a vector of these features. Suppose a house has 1 adjective in its ad, and the mortgage rate was 6.5 and there were 10,000 unsold houses in the city. The feature vector for the house would be $\vec{f} = (20000, 6.5, 10000)$. Suppose the weight vector that we had previously learned for this task was $\vec{w} = (w_0, w_1, w_2, w_3) = (18000, -5000, -3000, -1.8)$. Then the predicted value for this house would be computed by multiplying each feature by its weight:

(6.44) $$\text{price} = w_0 + \sum_{i=1}^{N} w_i \times f_i$$

In general we will pretend that there is an extra feature $f_0$ which has the value 1, an **intercept feature**, which makes the equations simpler with regard to that pesky $w_0$, and so in general we can represent a linear regression for estimating the value of $y$ as:

(6.45) $$\textbf{linear regression:} \qquad y = \sum_{i=0}^{N} w_i \times f_i$$

DOT PRODUCT

Taking two vectors and creating a scalar by multiplying each element in a pairwise fashion and summing the results is called the **dot product**. Recall that the dot product $a \cdot b$ between two vectors $a$ and $b$ is defined as:

(6.46) $$\textbf{dot product:} \qquad a \cdot b = \sum_{i=1}^{N} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Thus Eq. 6.45 is equivalent to the dot product between the weights vector and the feature vector:

(6.47) $$y = w \cdot f$$

Vector dot products occurs very frequently in speech and language processing; we will often rely on the dot product notation to avoid the messy summation signs.

### Learning in linear regression

How do we learn the weights for linear regression? Intuitively we'd like to choose weights that make the estimated values $y$ as close as possible to the actual values that we saw in the training set.

Consider a particular instance $x^{(j)}$ from the training set (we'll use superscripts in parentheses to represent training instances), with has an observed label in the training set $y_{obs}^{(j)}$. Our linear regression model predicts a value for $y^{(j)}$ as follows:

(6.48)
$$y^{(j)}_{pred} = \sum_{i=0}^{N} w_i \times f_i^{(j)}$$

We'd like to choose the whole set of weights $W$ so as to minimize the difference between the predicted value $y^{(j)}_{pred}$ and the observed value $y^{(j)}_{obs}$, and we want this difference minimized over all the $M$ examples in our training set. Actually we want to minimize the absolute value of the difference (since we don't want a negative distance in one example to cancel out a positive difference in another example), so for simplicity (and differentiability) we minimize the square of the difference. Thus the total value we want to minimize, which we call the **sum-squared error**, is this cost function of the current set of weights $W$:

SUM-SQUARED ERROR

(6.49)
$$\text{cost}(W) = \sum_{j=0}^{M} \left( y^{(j)}_{pred} - y^{(j)}_{obs} \right)^2$$

We won't give here the details of choosing the optimal set of weights to minimize the sum-squared error. But, briefly, it turns out that if we put the entire training set into a single matrix $X$ with each row in the matrix consisting of the vector of features associated with each observation $x^{(i)}$, and put all the observed $y$ values in a vector $\vec{y}$, that there is a closed-form formula for the optimal weight values $W$ which will minimize cost$(W)$:

(6.50)
$$W = (X^T X)^{-1} X^T \vec{y}$$

Implementations of this equation are widely available in statistical packages.

### 6.6.2 Logistic regression

Linear regression is what we want when we are predicting a real-valued outcome. But somewhat more commonly in speech and language processing we are doing **classification**, in which the output $y$ we are trying to predict takes on one from a small set of discrete values.

Consider the simplest case of binary classification, where we want to classify whether some observation $x$ is in the class (true) or not in the class (false). In other words $y$ can only take on the values 1 (true) or 0 (false), and we'd like a classifier that can take features of $x$ and return true or false. Furthermore, instead of just returning the 0 or 1 value, we'd like a model that can give us the **probability** that a particular observation is in class 0 or 1. This is important because in most real-world tasks we're passing the results of this classifier onto some further classifier to accomplish some task. Since we are rarely completely certain about which class an observation falls in, we'd prefer not to make a hard decision at this stage, ruling out all other classes. Instead, we'd like to pass on to the later classifier as much information as possible: the entire set of classes, with the probability value that we assign to each class.

Could we modify our linear regression model to use it for this kind of probabilistic classification? Suppose we just tried to train a linear model to predict a probability as follows:

$$P(y=true|x) = \sum_{i=0}^{N} w_i \times f_i \quad (6.51)$$

$$= w \cdot f \quad (6.52)$$

We could train such a model by assigning each training observation the target value $y=1$ if it was in the class (true) and the target value $y=0$ if it was not (false). Each observation $x$ would have a feature vector $f$, and we would train the weight vector $w$ to minimize the predictive error from 1 (for observations in the class) or 0 (for observations not in the class). After training, we would compute the probability of a class given an observation by just taking the dot product of the weight vector with the features for that observation.

The problem with this model is that there is nothing to force the output to be a legal probability, i.e. to lie between zero and 1. The expression $\sum_{i=0}^{N} w_i \times f_i$ produces values from $-\infty$ to $\infty$. How can we fix this problem? Suppose that we keep our linear predictor $w \cdot f$, but instead of having it predict a probability, we have it predict a *ratio* of two probabilities. Specifically, suppose we predict the ratio of the probability of being in the class to the probability of not being in the class. This ratio is called the **odds**; if an event has probability .75 of occurring and probability .25 of not occurring, we say the **odds** of occurring is $.75/.25 = 3$. We could use the linear model to predict the odds of $y$ being true:

ODDS

$$\frac{p(y=true)|x}{1-p(y=true|x)} = w \cdot f \quad (6.53)$$

This last model is close; a ratio of probabilities can lie between 0 and $\infty$. But we need the left-hand side of the equation to lie between $-\infty$ and $\infty$. We can achieve this by taking the natural log of this probability:

$$\ln\left(\frac{p(y=true|x)}{1-p(y=true|x)}\right) = w \cdot f \quad (6.54)$$

Now both the left and right hand lie between $-\infty$ and $\infty$. This function on the left (the log of the odds) is known as the **logit function**:

LOGIT FUNCTION

$$\text{logit}(p(x)) = \ln\left(\frac{p(x)}{1-p(x)}\right) \quad (6.55)$$

The model of regression in which we use a linear function to estimate, not the probability, but the logit of the probability, is known as **logistic regression**. If the linear function is estimating the logit, what is the actual formula in logistic regression for the probability $P(y=true)$? You should stop here and take Equation (6.54) and apply some simple algebra to solve for the probability $P(y=true)$.

LOGISTIC REGRESSION

Hopefully when you solved for $P(y=true)$ you came up with a derivation something like the following:

$$\ln\left(\frac{p(y=\text{true}|x)}{1-p(y=\text{true}|x)}\right) = w \cdot f$$

$$\frac{p(y=\text{true}|x)}{1-p(y=\text{true}|x)} = e^{w\cdot f} \tag{6.56}$$

$$p(y=\text{true}|x) = (1-p(y=\text{true}|x))e^{w\cdot f}$$

$$p(y=\text{true}|x) = e^{w\cdot f} - p(y=\text{true}|x)e^{w\cdot f}$$

$$p(y=\text{true}|x) + p(y=\text{true}|x)e^{w\cdot f} = e^{w\cdot f}$$

$$p(y=\text{true}|x)(1+e^{w\cdot f}) = e^{w\cdot f}$$

$$p(y=\text{true}|x) = \frac{e^{w\cdot f}}{1+e^{w\cdot f}} \tag{6.57}$$

Once we have this probability, we can easily state the probability of the observation not belonging to the class, $p(y=false|x)$, as the two must sum to 1:

$$p(y=false|x) = \frac{1}{1+e^{w\cdot f}} \tag{6.58}$$

Here are the equations again using explicit summation notation:

$$p(y=\text{true}|x) \;=\; \frac{\exp(\sum_{i=0}^{N} w_i f_i)}{1+\exp(\sum_{i=0}^{N} w_i f_i)} \tag{6.59}$$

$$p(y=\text{false}|x) \;=\; \frac{1}{1+\exp(\sum_{i=0}^{N} w_i f_i)} \tag{6.60}$$

We can express the probability $P(y=\text{true}|x)$ in a slightly different way, by dividing the numerator and denominator in (6.57) by $e^{-w\cdot f}$:

$$p(y=\text{true}|x) \;=\; \frac{e^{w\cdot f}}{1+e^{w\cdot f}} \tag{6.61}$$

$$\;=\; \frac{1}{1+e^{-w\cdot f}} \tag{6.62}$$

LOGISTIC FUNCTION

These last equation is now in the form of what is called the **logistic function**, (the function that gives logistic regression its name). The general form of the logistic function is:

$$\frac{1}{1+e^{-x}} \tag{6.63}$$

Fig. 6.18 shows a picture of the logistic function; note how values from $-\infty$ and $\infty$ are neatly mapped to lie between 0 and 1.

Again, we can express $P(y=\text{false}|x)$ so as to make the probabilities sum to one:

$$p(y=\text{false}|x) \;=\; \frac{e^{-w\cdot f}}{1+e^{-w\cdot f}} \tag{6.64}$$

**Figure 6.18** FIX: ADD PICTURE OF LOGISTIC FUNCTION HERE

### 6.6.3 Logistic regression: Classification

CLASSIFICATION

INFERENCE

Given a particular observation, how do we decide which of the two classes ('true' or 'false') it belongs to? This is the task of **classification**, also called **inference**. Clearly the correct class is the one with the higher probability. Thus we can safely say that our observation should be labeled 'true' if:

$$\begin{aligned} p(y=true|x) &> p(y=false|x) \\ \frac{p(y=true|x)}{p(y=false|x)} &> 1 \\ \frac{p(y=true|x)}{1-p(y=true|x)} &> 1 \end{aligned}$$

and substituting from Eq. 6.56 for the odds ratio:

$$\begin{aligned} e^{w\cdot f} &> 1 \\ w\cdot f &> 0 \end{aligned} \tag{6.65}$$

or with the explicit sum notation:

$$\sum_{i=0}^{N} w_i f_i > 0 \tag{6.66}$$

Thus in order to decide if an observation is a member of the class we just need to compute the linear function, and see if its value is positive; if so, the observation is in the class.

A more advanced point: the equation $\sum_{i=0}^{N} w_i f_i = 0$ is the equation of a hyperplane (a generalization of a line to $N$ dimensions). The equation $\sum_{i=0}^{N} w_i f_i > 0$ is thus the part of $N$-dimensional space above this hyperplane. Thus we can see the logistic regression function as learning a hyperplane which separates points in space which are in the class ('true') from points which are not in the class. Fig. 6.19 shows a graphic intuition of this hyperplane.

**Figure 6.19** FIX: ADD PICTURE OF A HYPERPLANE SEPARATING CLOUD OF POINTS HERE

### 6.6.4 Advanced: Learning in logistic regression

CONDITIONAL MAXIMIUM LIKELIHOOD ESTIMATION

In linear regression, learning consisted of choosing the weights $w$ which minimized the sum-squared error on the training set. In logistic regression, by contrast, we generally use **conditional maximium likelihood estimation**. What this means is that we choose the parameters $w$ which makes the probability of the observed $y$ values in the training data to be the highest, given the observations $x$. In other words, for an individual training observation $x$, we want to choose the weights as follows:

$$\hat{w} = \underset{w}{\operatorname{argmax}} P(y^{(i)}|x^{(i)}) \tag{6.67}$$

And we'd like to choose the optimal weights for the entire training set:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \prod_i P(y^{(i)}|x^{(i)}) \tag{6.68}$$

We generally work with the log likelihood:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_i \log P(y^{(i)}|x^{(i)}) \tag{6.69}$$

So, more explicitly:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_i \log \begin{cases} P(y^{(i)} = 1|x^{(i)})) & \text{for } y^{(i)} = 1 \\ P(y^{(i)} = 0|x^{(i)})) & \text{for } y^{(i)} = 0 \end{cases} \tag{6.70}$$

This equation is unwieldy, and so we usually apply a convenient representational trick; note that if $y = 0$ the first term goes away, while if $y = 1$ the second term goes away:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_i y^{(i)} \log P(y^{(i)} = 1|x^{(i)})) + (1 - y^{(i)}) \log P(y^{(i)} = 0|x^{(i)}) \tag{6.71}$$

Now if we substitute in (6.62) and (6.64), we get:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_i y^{(i)} \log \frac{e^{-w\cdot f}}{1+e^{-w\cdot f}} + (1 - y^{(i)}) \log \frac{1}{1+e^{-w\cdot f}} \tag{6.72}$$

CONVEX OPTIMIZATION

Finding the weights which result in the maximum log-likelihood according to (6.72) is a problem in the field known as **convex optimization**. Among the most commonly used algorithms **quasi-Newton** methods like L-BFGS, as well as gradient ascent, conjugate gradient, and various iterative scaling algorithms (?; Pietra et al., 1997; Malouf, 2002). These learning algorithms are available in the various MaxEnt modeling toolkits but are too complex to define here; interested readers should see the machine learning textbooks suggested at the end of the chapter.

## 6.7 Maximum Entropy Modeling

We showed above how logistic regression can be used to classify an observation into one of two classes. But most of the time the kinds of classification problems that come up in language processing involve larger numbers of classes (such as the set of part-of-speech classes). Logistic regression can also be defined for such functions with

MULTINOMIAL LOGISTIC REGRESSION
MAXENT

many discrete values. In such cases it is called **multinomial logistic regression**. As we mentioned above, multinomial logistic regression is called **MaxEnt** in speech and language processing (see Sec. 6.7.1 for the intuition for the name 'maximum entropy').

The equations for computing the class probabilities for a maxent classifier are a generalization of Eqs. 6.59-6.60 above. Let's assume that the target value $y$ is a random variable which can take on $C$ different values corresponding to the classes $c_1$, $c_2$,...,$c_C$.

We said earlier in this chapter that in a MaxEnt model we estimate the probability that $y$ is a particular class $c$ as:

$$p(c|x) \;=\; \frac{1}{Z}\exp\sum_i w_i f_i \tag{6.73}$$

Let's now add some details to this schematic equation. First we'll flesh out the normalization factor Z, specify the number of features as $N$, and make the value of the weight dependent on the class $c$; the final equation is thus:

$$p(c|x) \;=\; \frac{\exp\left(\sum_{i=0}^{N} w_{ci} f_i\right)}{\sum_{c'\in C}\exp\left(\sum_{i=0}^{N} w_{c'i} f_i\right)} \tag{6.74}$$

Note that the normalization factor $Z$ is just used to make the exponential into a true probability;

$$Z = \sum_C p(c|x) = \sum_{c'\in C}\exp\left(\sum_{i=0}^{N} w_{c'i} f_i\right) \tag{6.75}$$

We need to make one more change to see the final MaxEnt equation. So far we've been assuming that the features $f_i$ are real-valued. It is more common in speech and language processing, however, to use binary-valued features. A feature that only takes on the values 0 and 1 is also called an **indicator function**. In general, the features we use are indicator functions of some property of the observation and the class we are considering assigning. Thus in MaxEnt, instead of the notation $f_i$, we will often use the notation $f_i(c,x)$, meaning that a feature $i$ for a particular class $c$ for a given observation $x$.

INDICATOR FUNCTION

The final equation for computing the probability of $y$ being class $c$ given $x$ in MaxEnt is:

$$p(c|x) \;=\; \frac{\exp\left(\sum_{i=0}^{N} w_{ci} f_i(c,x)\right)}{\sum_{c'\in C}\exp\left(\sum_{i=0}^{N} w_{c'i} f_i(c',x)\right)} \tag{6.76}$$

To get a clearer intuition of this use of binary features, let's look at some sample features for the task of part-of-speech tagging. Suppose we are assigning a part-of-speech tag to the word *race* in (6.77), repeated from (**??**):

(6.77) Secretariat/NNP is/BEZ expected/VBN to/TO **race**/??︎ tomorrow/

Again, for now we're just doing classification, not sequence classification, so let's consider just this single word; we'll discuss in Sec. 6.8 how to perform tagging for a whole sequence of words.

We would like to know whether to assign the class *VB* to *race* (or instead assign some other class like *NN*). One useful feature, we'll call it $f_1$, would be the fact that the current word is *race*. We can thus add a binary feature which is true if this is the case:

$$f_1(c,x) \;=\; \begin{cases} 1 & \text{if } word_i = \text{“race”} \;\;\&\;\; c = \text{NN} \\ 0 & \text{otherwise} \end{cases}$$

Another feature would be whether the previous word has the tag *TO*:

$$f_2(c,x) \;=\; \begin{cases} 1 & \text{if } t_{i-1} = \text{TO} \;\;\&\;\; c = \text{VB} \\ 0 & \text{otherwise} \end{cases}$$

Two more part-of-speech tagging features might focus on aspects of a word's spelling and case:

$$f_3(c,x) \;=\; \begin{cases} 1 & \text{if } \text{suffix}(word_i) = \text{“ing”} \;\;\&\;\; c = \text{VBG} \\ 0 & \text{otherwise} \end{cases}$$

$$f_4(c,x) \;=\; \begin{cases} 1 & \text{if } \text{is\_lower\_case}(word_i) \;\;\&\;\; c = \text{VB} \\ 0 & \text{otherwise} \end{cases}$$

Since each feature is dependent on both a property of the observation and the class being labeled, we would need to have separate feature for, e.g, the link between *race* and VB, or the link between a previous TO and NN:

$$f_5(c,x) \;=\; \begin{cases} 1 & \text{if } word_i = \text{”race”} \;\;\&\;\; c = \text{VB} \\ 0 & \text{otherwise} \end{cases}$$

$$f_6(c,x) \;=\; \begin{cases} 1 & \text{if } t_{i-1} = \text{TO} \;\;\&\;\; c = \text{NN} \\ 0 & \text{otherwise} \end{cases}$$

Each of these features has a corresponding weight. Thus the weight $w_1(c,x)$ would indicate how strong a cue the word *race* is for the tag VB, the weight $w_2(c,x)$ would indicate how strong a cue the previous tag *TO* is for the current word being a VB, and so on.

| | | f1 | f2 | f3 | f4 | f5 | f6 |
|---|---|---|---|---|---|---|---|
| VB | f | 0 | 1 | 0 | 1 | 1 | 0 |
| VB | w | | .8 | | .01 | .1 | |
| NN | f | 1 | 0 | 0 | 0 | 0 | 1 |
| NN | w | .8 | | | | | -1.3 |

**Figure 6.20** Some sample feature values and weights for tagging the word *race* in (6.77).

Let's assume that the feature weights for the two class VB and VN are as shown in Fig. 6.20. Let's call the current input observation (where the current word is *race*) $x$. We can now compute $P(NN|x)$ and $P(VB|x)$, using Eq. 6.76:

$$P(NN|x) = \frac{e^{.8}e^{-1.3}}{e^{.8}e^{-1.3}+e^{.8}e^{.01}e^{.1}} = .20 \tag{6.78}$$

$$P(VB|x) = \frac{e^{.8}e^{.01}e^{.1}}{e^{.8}e^{-1.3}+e^{.8}e^{.01}e^{.1}} = .80 \tag{6.79}$$

Notice that when we use MaxEnt to perform **classification**, MaxEnt naturally gives us a probability distribution over the classes. If we want to do a hard-classification and choose the single-best class, we can choose the class that has the highest probability, i.e.:

$$\hat{c} = \operatorname*{argmax}_{c \in C} P(c|x) \tag{6.80}$$

Classification in MaxEnt is thus a generalization of classification in (boolean) logistic regression. In boolean logistic regression, classification involves building one linear expression which separates the observations in the class from the observations not in the class. Classification in MaxEnt, by contrast, involves building a separate linear expression for each of $C$ classes.

But as we'll see later in Sec. 6.8, we generally don't use MaxEnt for hard classification. Usually we want to use MaxEnt as part of sequence classification, where we want, not the best single class for one unit, but the best total sequence. For this task, it's useful to exploit the entire probability distribution for each individual unit, to help find the best sequence. Indeed even in many non-sequence applications a probability distribution over the classes is more useful than a hard choice.

The features we have described so far express a single binary property of an observation. But it is often useful to create more complex features that express combinations of properties of a word. Some kinds of machine learning models, like Support Vector Machines (SVMs), automatically model the interactions between primitive properties, but in MaxEnt any kind of complex feature has to be defined by hand. For example a word starting with a capital letter (like the word *Day* is more likely to be a proper noun (NNP) than a common noun (for example in the expression *United Nations Day*. But a word which is capitalized but which occurs at the beginning of the sentence (the previous word is `</s>`), as in *Day after day....*, is not more likely to be a proper noun. Even if each of these properties were already a primitive feature, MaxEnt would not model their combination, so this boolean combination of properties would need to be encoded as a feature by hand:

$$f_{125}(c,x) \;=\; \begin{cases} 1 & \text{if } word_{i-1} = \texttt{<s>} \;\&\; \text{isupperfirst}(word_i) \;\&\; c = \text{NNP} \\ 0 & \text{otherwise} \end{cases}$$

A key to successful use of MaxEnt is thus the design of appropriate features and feature combinations.

### Learning Maximum Entropy Models

Learning a MaxEnt model can be done via a generalization of the logistic regression learning algorithms described in Sec. 6.6.4; as we saw in (6.69), we want to find the parameters $w$ which maximize the likelihood of the $M$ training samples:

$$\hat{w} = \operatorname*{argmax}_{w} \prod_{i}^{M} P(y^{(i)}|x^{(i)}) \tag{6.81}$$

As with binary logistic regression, we use some convex optimization algorithm to find the weights which maximize this function.

A brief note: one important aspect of MaxEnt training is a kind of smoothing of the weights called **regularization**. The goal of regularization is to penalize large weights; it turns out that otherwise, a MaxEnt model will learn very high weights which overfit the training data. Regularization is implemented in training by changing the likelihood function that is optimized. Instead of the optimization in (6.81), we optimize the following:

REGULARIZATION

$$\hat{w} = \operatorname*{argmax}_{w} \sum_{i} \log P(y^{(i)}|x^{(i)}) - \alpha R(w) \tag{6.82}$$

where $R(w)$ is a **regularization** term used to penalize large weights. It is common to make the regularization term $R(w)$ be a quadratic function of the weight values:

$$R(W) = \sum_{j=1}^{N} w_j^2 \tag{6.83}$$

Subtracting squares of the weights will thus result in preferring smaller weights:

$$\hat{w} = \operatorname*{argmax}_{w} \sum_{i} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{N} w_j^2 \tag{6.84}$$

It turns that this kind of regularization corresponds to assuming that weights are distributed according to a Gaussian distribution with mean $\mu = 0$. In a Gaussian or normal distribution, the further away a value is from the mean, the lower its probability (scaled by the variance $\sigma$). By using a Gaussian prior on the weights, we are saying that weights prefer to have the value zero. A Gaussian for a weight $w_j$ is:

$$\frac{1}{\sqrt{2\pi\sigma_j^2}}\exp\left(-\frac{(w_j-\mu_j)^2}{2\sigma_j^2}\right) \tag{6.85}$$

If we multiply each weight by a Gaussian prior on the weight, we are thus maximizing the following constraint:

$$\hat{w} = \operatorname*{argmax}_w \prod_i^M P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^N \frac{1}{\sqrt{2\pi\sigma_j^2}}\exp\left(-\frac{(w_j-\mu_j)^2}{2\sigma_j^2}\right) \tag{6.86}$$

which in log space, with $\mu = 0$, corresponds to

$$\hat{w} = \operatorname*{argmax}_w \sum_i \log P(y^{(i)}|x^{(i)}) - \sum_{j=1}^N \frac{w_j^2}{2\sigma_j^2} \tag{6.87}$$

which is in the same form as Eq. 6.84.

There is a vast literature on the details of learning in MaxEnt; see the end of the chapter for pointers to further details.

### 6.7.1 Why do we call it Maximum Entropy?

Why do we refer to multinomial logistic regression models as MaxEnt or Maximum Entropy models? Let's give the intuition of this interpretation in the context of part-of-speech tagging. Suppose we want to assign a tag to the word *zzfish* (a word we made up for this example). What is the probabilistic tagging model (the distribution of part-of-speech tags across words) that makes the fewest assumptions, imposing no constraints at all? Intuitively it would be the equiprobable distribution:

| NN | JJ | NNS | VB | NNP | IN | MD | UH | SYM | VBG | POS | PRP | CC | CD | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | $\frac{1}{45}$ | ... |

Now suppose we had some training data labeled with part-of-speech tags, and from this data we learned only one fact: the set of possible tags for *zzfish* are NN, JJ, NNS, and VB (so *zzfish* is a word something like *fish*, but which can also be an adjective). What is the tagging model which relies on this constraint, but makes no further assumptions at all? Since one of these must be the correct tag, we know that

$$P(NN)+P(JJ)+P(NNS)+P(VB) = 1 \tag{6.88}$$

Since we have no further information, a model which makes no further assumptions beyond what we know would simply assign equal probability to each of these words:

| NN | JJ | NNS | VB | NNP | IN | MD | UH | SYM | VBG | POS | PRP | CC | CD | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |

In the first example, where we wanted an uninformed distribution over 45 parts-of-speech, and in this case, where we wanted an uninformed distribution over 4 parts-of-speech, it turns out that of all possible distributions, the equiprobable distribution has the **maximum entropy**. Recall from Sec. **??** that the entropy of the distribution of a random variable $x$ is computed as:

$$H(x) = -\sum_{x} P(x)\log_2 P(x) \tag{6.89}$$

An equiprobable distribution in which all values of the random variable have the same probability has a higher entropy than one in which there is more information. Thus of all distributions over four variables the distribution $\{\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\}$ has the maximum entropy. (To have an intuition for this, use Eq. 6.89 to compute the entropy for a few other distributions such as the distribution $\{\frac{1}{4},\frac{1}{2},\frac{1}{8},\frac{1}{8}\}$, and make sure they are all lower than the equiprobable distribution.)

The intuition of MaxEnt modeling is that the probabilistic model we are building should follow whatever constraints we impose on it, but beyond these constraints it should follow Occam's Razor, i.e. make the fewest possible assumptions.

Let's add some more constraints into our tagging example. Suppose we looked at our tagged training data and noticed that 8 times out of 10, *zzfish* was tagged as some sort of common noun, either NN or NNS. We can think of this as specifying the feature 'word is *zzfish* and $t_i$ = NN or $t_i$ = NNS'. We might now want to modify our distribution so that we give $\frac{8}{10}$ of our probability mass to nouns, i.e. now we have 2 constraints

$$\begin{aligned} P(NN)+P(JJ)+P(NNS)+P(VB) &= 1 \\ P(\text{word is } \textit{zzfish} \text{ and } t_i = \text{NN or } t_i = \text{NNS}) &= \frac{8}{10} \end{aligned}$$

but make no further assumptions (keep JJ and VB equiprobable, and NN and NNS equiprobable).

| NN | JJ | NNS | VB | NNP | ... |
|---|---|---|---|---|---|
| $\frac{4}{10}$ | $\frac{1}{10}$ | $\frac{4}{10}$ | $\frac{1}{10}$ | 0 | ... |

Now suppose we don't have have any more information about *zzfish*. But we notice in the training data that for all English words (not just *zzfish*) that verbs (VB) occur as 1 word in 20. We can now add this constraint (corresponding to the feature $t_i = VB$):

$$\begin{aligned} &P(NN)+P(JJ)+P(NNS)+P(VB) = 1 \\ &P(\text{word is } \textit{zzfish} \text{ and } t_i = \text{NN or } t_i = \text{NNS}) = \frac{6}{10} \\ &P(VB) = \frac{1}{20} \end{aligned}$$

The resulting maximum entropy distribution is now as follows:

| NN | JJ | NNS | VB |
|---|---|---|---|
| $\frac{4}{10}$ | $\frac{3}{20}$ | $\frac{4}{10}$ | $\frac{1}{20}$ |

In summary, the intuition of maximum entropy is to build a distribution by continuously adding features. Each feature is an indicator function, which picks out a subset of the training observations. For each feature we add a constraint on our total distribution, specifying that our distribution for this subset should match the empirical distribution we saw in our training data. We then choose the maximum entropy distribution which otherwise accords with these constraints. Berger et al. (1996) pose the optimization problem of finding this distribution as follows:

> *"To select a model from a set $C$ of allowed probability distributions, choose the model $p_* \in C$ with maximum entropy $H(p)$":*

$$p_* = \underset{p \in C}{\operatorname{argmax}} H(p) \tag{6.90}$$

Now we come to the important conclusion. Berger et al. (1996) show that the solution to this optimization problem turns out to be exactly the probability distribution of a multinomial logistic regression model whose weights $W$ maximize the likelihood of the training data! Thus the exponential model for multinomial logistic regression, when trained according to the maximum likelihood criterion, also finds the maximum entropy distribution subject to the constraints from the feature functions.

## 6.8 Maximum Entropy Markov Models

We began our discussion of MaxEnt by pointing out that the basic MaxEnt model is not in itself a classifier for sequences. Instead, it is used to classify a single observation into one of a set of discrete classes, as in text classification (choosing between possible authors of an anonymous text, or classifying an email as spam), or tasks like deciding whether a period marks the end of a sentence.

We turn in this section to the **Maximum Entropy Markov Model** or **MEMM**, which is an augmentation of the basic MaxEnt classifier so that it can be applied to assign a class to each element in a sequence, just as we do with HMMs. Why would we want a sequence classifier built on MaxEnt? How might such a classifier be better than an HMM?

Consider the HMM approach to part-of-speech tagging. The HMM tagging model is based on probabilities of the form $P(\text{tag}|\text{tag})$ and $P(\text{word}|\text{tag})$. That means that 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. But many knowledge sources are hard to fit into these models. For example, we saw in Sec. **??** that for tagging unknown words, useful features include capitalization, the presence of hyphens, word endings, and so on. There is no easy way to fit these features into an HMM-style model; as we discussed in Ch. 5, $P(\text{capitalization}|\text{tag})$, $P(\text{hyphen}|\text{tag})$, $P(\text{suffix}|\text{tag})$, and so on

We gave the initial part of this intuition in the previous section, when we discussed applying MaxEnt to part-of-speech tagging. Part-of-speech tagging is definitely a sequence-labeling task, but we only discussed assigning a part-of-speech tag to a single word.

How can we take this single local classifier and turn it into a general sequence classifier? When classifying each word we can rely on features from the current word, features from surrounding words, as well as the output of the classifier from previous words. For example the simplest method is to run our local classifier left-to-right, first making a hard classification of the first word in the sentence, then the second word, and so on. When classifying each word, we can rely on the output of the classifier from the previous word as a feature. For example, we saw in tagging the word *race* that a useful feature was the tag of the previous word; a previous TO is a good indication that *race* is a VB, whereas a previous DT is a good indication that *race* is a NN. Such a strict left-to-right sliding window approach has been shown to yield surprisingly good results across a wide range of applications.

While it is possible to perform part-of-speech tagging in this way, this simple left-to-right classifier has an important flaw: it makes a hard decision on each word before moving on to the next word. This means that the classifier is unable to use information from later words to inform its decision early on. Recall that in Hidden Markov Models, by contrast, we didn't have to make a hard decision at each word; we used Viterbi decoding to find the sequence of part-of-speech tags which was optimal for the whole sentence.

The Maximum Entropy Markov Model (or MEMM) allows us to achieve this same advantage, by mating the Viterbi algorithm with MaxEnt. Let's see how it works, again looking at part-of-speech tagging. It is easiest to understand an MEMM when comparing it to an HMM. Remember that in using an HMM to model the most probable part-of-speech tag sequence we rely on Bayes rule, computing $P(W|T)P(W)$ instead of directly computing $P(T|W)$:

$$\begin{aligned}
\hat{T} &= \operatorname*{argmax}_T P(T|W) \\
&= \operatorname*{argmax}_T P(W|T)P(T) \\
&= \operatorname*{argmax}_T \prod_i P(word_i|tag_i) \prod_i P(tag_i|tag_{i-1}) \qquad (6.91)
\end{aligned}$$

That is, an HMM as we've described it is a generative model, which optimizes the likelihood $P(W|T)$, and we estimate the posterior by combining the likelihood and the prior $P(T)$.

In an MEMM, by contrast, we compute the posterior $P(T|W)$ directly. Because we train the model directly to discriminate among the possible tag sequences, we call an MEMM a **discriminative** rather than generative model. In an MEMM, we break down the probabilities as follows:

DISCRIMINATIVE

$$\begin{aligned}
\hat{T} &= \operatorname*{argmax}_T P(T|W) \\
&= \operatorname*{argmax}_T \prod_i P(tag_i|word_i, tag_{i-1}) \qquad (6.92)
\end{aligned}$$

Thus in an MEMM instead of having a separate model for likelihoods and priors, we train a single probabilistic model to estimate $P(tag_i|word_i, tag_{i-1})$. We will use MaxEnt for this last piece, estimating the probability of each local tag given the previous tag, the observed word, and, as we will see, any other features we want to include.

We can see the HMM versus MEMM intuitions of the POS tagging task in Fig. 6.21, which repeats the HMM model of Fig. **??**a, and adds a new model for the MEMM. Note that the HMM model includes distinct probability estimates for each transition and observation, while the MEMM gives one probability estimate per hidden state, which is the probability of the next tag given the previous tag and the observation.

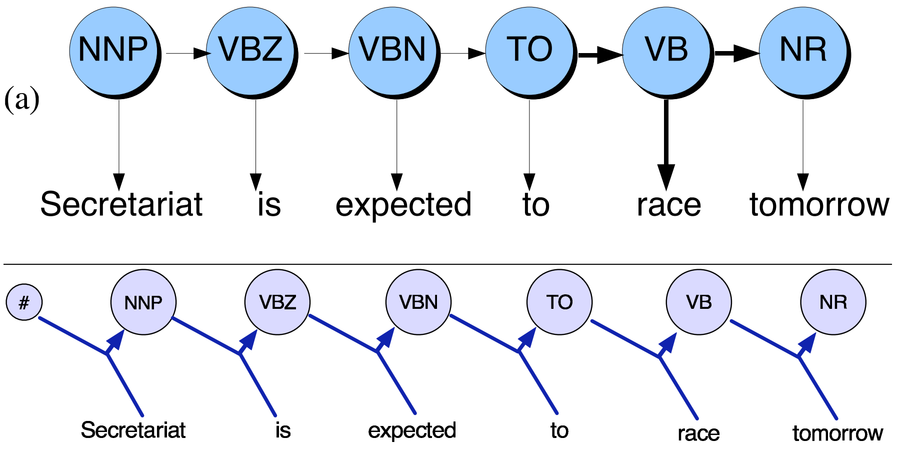

**Figure 6.21** The HMM (top) and MEMM (bottom) representation of the probability computation for the correct sequence of tags for the Secretariat sentence. Each arc would be associated with a probability; the HMM computes two separate probabilities for the observation likelihood and the prior, while the MEMM computes a single probability function at each state, conditioned on the previous state and observation.

Fig. 6.22 emphasizes another advantage of MEMMs over HMMs not shown in Fig. 6.21: unlike the HMM, the MEMM can condition on any useful feature of the input observation. In the HMM this wasn't possible because the HMM is likelihood-based, and hence would have needed compute the likelihood of each feature of the observation.

More formally, in the HMM we compute the probability of the state sequence given the observations as:

$$P(Q|O) = \prod_{i=1}^{n} P(o_i|q_i) \times \prod_{i=1}^{n} P(q_i|q_{i-1}) \tag{6.93}$$

In the MEMM, we compute the probability of the state sequence given the observations as:

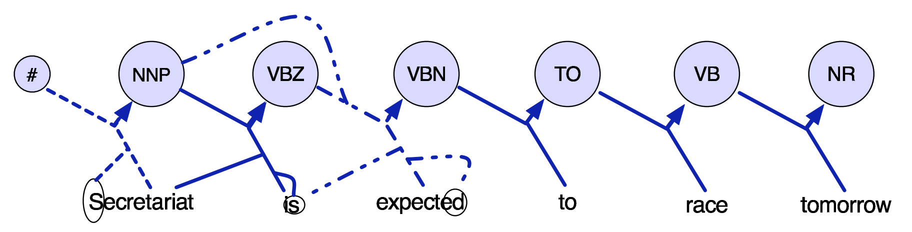

**Figure 6.22** An MEMM for part-of-speech tagging, augmenting the description in Fig. 6.21 by showing that an MEMM can condition on many features of the input, such as capitalization, morphology (ending in *-s* or *-ed*), as well as earlier words or tags. We have shown some potential additional features for the first three decisions, using different line styles for each class for clarity.

$$P(Q|O) = \prod_{i=1}^{n} P(q_i|q_{i-1}, o_i) \tag{6.94}$$

In practice, however, an MEMM can condition on many more features than the HMM as well, so in general we condition the right-hand side on many more factors.

To estimate the individual probability of a transition from a state $q'$ to a state $q$ producing an observation $o$, we build a MaxEnt model as follows:

$$P(q|q', o) = \frac{1}{Z(o, q')} \exp\left(\sum_i w_i f_i(o, q)\right) \tag{6.95}$$

### 6.8.1 Decoding and Learning in MEMMs

Like HMMs, the MEMM uses the Viterbi algorithm to perform the task of decoding (inference). Concretely, this involves filling an $N \times T$ array with the appropriate values for $P(t_i|t_{i-1}, word_i)$, maintaining backpointers as we proceed. As with the 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 are isolated to how we fill each cell. Recall from Eq˙ **??** that the recursive step of the Viterbi equation computes the Viterbi value of time $t$ for state $j$ as:

$$v_t(j) \;=\; \max_{1 \le i \le N-1} \; v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 < j < N, 1 < t < T \tag{6.96}$$

which is the HMM implementation of

$$v_t(j) \;=\; \max_{1 \le i \le N-1} \; v_{t-1}(i)\, P(s_j|s_i)\, P(o_t|s_j); \quad 1 < j < N, 1 < t < T \tag{6.97}$$

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

$$v_t(j) \;=\; \max_{1\leq i\leq N-1} \; v_{t-1}(i)\, P(s_j|s_i,o_t); \quad 1<j<N, 1<t<T \tag{6.98}$$

Fig. 6.23 shows an example of the Viterbi trellis for an MEMM applied to the ice-cream task from Sec. 6.4. Recall that the task is figuring out the hidden weather (Hot or Cold) from observed numbers of ice-cream eatings in Jason Eisner's diary. Fig. 6.23 shows the abstract Viterbi probability calculation assuming that we have a MaxEnt model which computes $P(s_i|s_{i-1},o_i)$ for us.

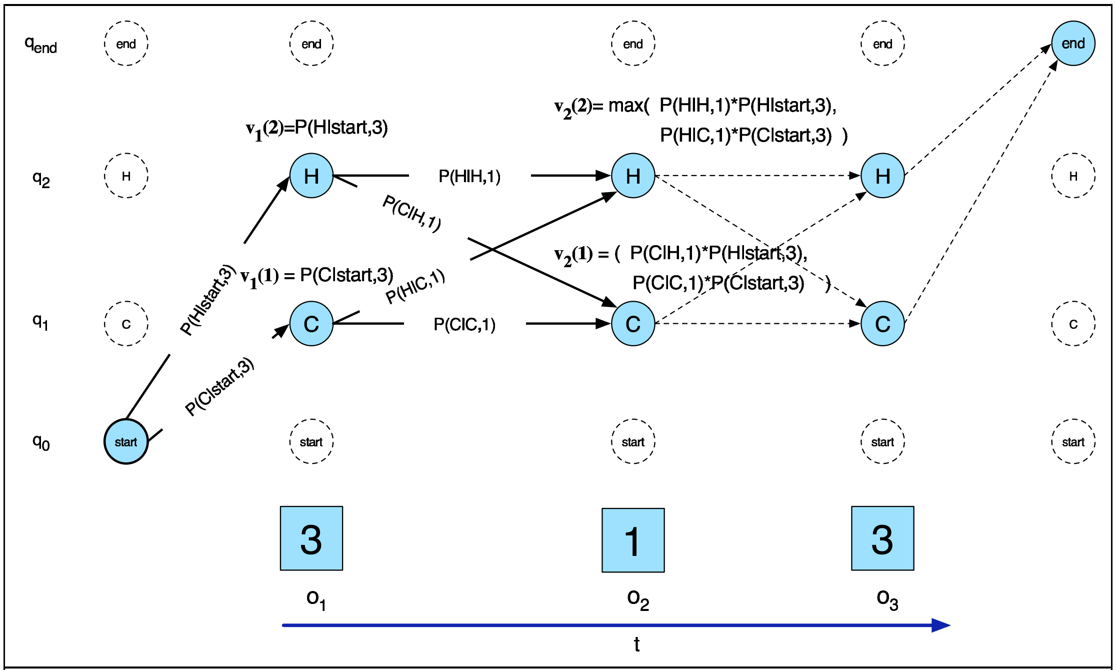

**Figure 6.23** Inference from ice-cream eating computed by an MEMM instead of an HMM. The Viterbi trellis for computing the best path through the hidden state space for the ice-cream eating events *3 1 3*, modified from the HMM figure in Fig. 6.9.

Learning in MEMMs relies on the same supervised learning algorithms we presented for logistic regression and MaxEnt. 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 HMMs, it is also possible to train MEMMs in semi-supervised modes, for example when the sequence of labels for the training data is missing or incomplete in some way; a version of the EM algorithm can be used for this purpose.

## 6.9 Summary

This chapter described two important models for probabilistic **sequence classification**, the **Hidden Markov Model** and the **Maximum Entropy Markov Model**. Both models are widely used throughout speech and language processing.

- Hidden Markov Models (**HMMs**) are a way of relating a sequence of **observations** to a sequence of **hidden classes** or **hidden states** which 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 using the **Baum-Welch** or **forward-backward** algorithm.
- A **MaxEnt** model is a classifier which assigns a **class** to an **observation** by computing a probability from an exponential function of a **weighted** set of **features** of the observation.
- MaxEnt models can be trained using methods from the field of **convex optimization** although we don't give the details in this textbook.
- A **Maximum Entropy Markov Model** or **MEMM** is a sequence model augmentation of MaxEnt which makes use of the Viterbi decoding algorithm.
- MEMMs can be trained by augmenting MaxEnt training with a version of EM.

## Bibliographical and Historical Notes

As we discussed at the end of Ch. 4, Markov chains were first used by Markov (1913), 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'[2] Kruskal and others give at least the following independently-discovered variants of the algorithm published in four separate fields:

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

| 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) is an early survey paper which 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) or Rabiner and Juang (1993) give very complete descriptions of the 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). ? (?) is a tutorial on EM.

While logistic regression and other log-linear models had been used in many fields since the middle of the 20th century, the use of Maximum Entropy/multinomial logistic regression in natural language processing dates from work in the early 1990s at IBM (Berger et al., 1996; Pietra et al., 1997). This early work introduced the maximum entropy formalism, proposed a learning algorithm (improved iterative scaling), and proposed the use of regularization. A number of applications of MaxEnt followed. For example for further discussion of regularization and smoothing for maximum entropy models see (*inter alia*) Chen and Rosenfeld (2000), Goodman (2004), and Dud́ik and Schapire (2006).

Although the second part of this chapter focused on MaxEnt-style classification, numerous other approaches to classification are used throughout speech and language processing. Naive Bayes (Duda et al., 2000) is often employed as a good baseline method (often yielding results that are sufficiently good for practical use; we'll cover naive Bayes in Ch. 19. Support Vector Machines (?) have been successfully used in text classification and in a wide variety of sequence processing applications. Decision lists ? (?)) have been widely used in word sense discrimination, and decision trees Breiman et al. (1984), Quinlan (1986) have been used in many applications in speech processing. Good references to supervised machine learning approaches to classification include Duda et al. (2000), ? (?), and Witten and Frank (2005).

Maximum Entropy Markov Models (MEMMs) were introduced by Ratnaparkhi (1996) and McCallum et al. (2000).

There are many sequence models that augment the MEMM. Conditional Random Field (CRF), various max-margin sequence classifiers...

Baum, L. E. (1972). An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. In Shisha, O. (Ed.), *Inequalities III: Proceedings of the Third Symposium on Inequalities*, University of California, Los Angeles, pp. 1–8. Academic Press.

Baum, L. E. and Eagon, J. A. (1967). An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bulletin of the American Mathematical Society*, *73*(3), 360–363.

Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite-state Markov chains. *Annals of Mathematical Statistics*, *37*(6), 1554–1563.

Berger, A. L., Pietra, S. A. D., and Pietra, V. J. D. (1996). A maximum entropy approach to natural language processing. *Computational Linguistics*, *22*(1), 39–71.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth & Brooks, Pacific Grove, California.

Chen, S. F. and Rosenfeld, R. (2000). A survey of smoothing techniques for ME models. *IEEE Transactions on Speech and Audio Processing*, *8*(1), 37–50.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the *EM* algorithm. *Journal of the Royal Statistical Society*, *39*(1), 1–21.

Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification*. Wiley-Interscience Publication.

Dud́ik, M. and Schapire, R. E. (2006). Maximum entropy distribution estimation with generalized regularization. In Lugosi, G. and Simon, H. (Eds.), *COLT 2006*, Berlin, pp. 123–138. Springer-Verlag.

Eisner, J. (2002). An interactive spreadsheet for teaching the forward-backward algorithm. In *Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL*, pp. 10–18.

Forney, Jr., G. D. (1973). The Viterbi algorithm. *Proceedings of the IEEE*, *61*(3), 268–278.

Goodman, J. (2004). Exponential priors for maximum entropy models. In *Proceedings of ACL-04*.

Jelinek, F. (1997). *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA.

Kruskal, J. B. (1983). An overview of sequence comparison. In Sankoff, D. and Kruskal, J. B. (Eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 1–44. Addison-Wesley, Reading, MA.

Levitt, S. D. and Dubner, S. J. (2005). *Freakonomics*. Morrow.

Malouf, R. (2002). A comparison of algorithms for maximum entropy parameter estimation. In *CoNNL-2002*, pp. 49–55.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.

Markov, A. A. (1913). Essai d'une recherche statistique sur le texte du roman "Eugene Onegin" illustrant la liaison des epreuve en chain ('Example of a statistical investigation of the text of "Eugene Onegin" illustrating the dependence between samples in chain'). *Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St.-Pétersbourg)*, *7*, 153–162. English translation by Morris Halle, 1956.

McCallum, A., Freitag, D., and Pereira, F. C. N. (2000). Maximum Entropy Markov Models for Information Extraction and Segmentation. In *ICML 2000*, pp. 591–598.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology*, *48*, 443–453.

Pietra, S. D., Pietra, V. J. D., and Lafferty, J. D. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(4), 380–393.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Rabiner, L. R. (1989). A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Rabiner, L. R. and Juang, B. (1993). *Fundamentals of Speech Recognition*. Prentice Hall, Englewood Cliffs, NJ.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, University of Pennsylvania, pp. 133–142. ACL.

Reichert, T. A., Cohen, D. N., and Wong, A. K. C. (1973). An application of information theory to genetic mutations and the matching of polypeptide sequences. *Journal of Theoretical Biology*, *42*, 245–261.

Sakoe, H. and Chiba, S. (1971). A dynamic programming approach to continuous speech recognition. In *Proceedings of the Seventh International Congress on Acoustics, Budapest*, Budapest, Vol. 3, pp. 65–69. Akadémiai Kiadó.

Sankoff, D. (1972). Matching sequences under deletion-insertion constraints. *Proceedings of the Natural Academy of Sciences of the U.S.A.*, *69*, 4–6.

Vintsyuk, T. K. (1968). Speech discrimination by dynamic programming. *Cybernetics*, *4*(1), 52–57. Russian Kibernetika 4(1):81-88 (1968).

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, *IT-13*(2), 260–269.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the Association for Computing Machinery*, *21*, 168–173.

Witten, I. H. and Frank, E. (2005). *Data Mining:: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann. 2nd ed.