17 *Markov and hidden Markov models*

17.1 Introduction

In this chapter, we discuss probabilistic models for sequences of observations, $X_1, \ldots, X_T$, of arbitrary length T. Such models have applications in computational biology, natural language processing, time series forecasting, etc. We focus on the case where we the observations occur at discrete "time steps", although "time" may also refer to locations within a sequence.

17.2 Markov models

Recall from Section 10.2.2 that the basic idea behind a Markov chain is to assume that X_t captures all the relevant information for predicting the future (i.e., we assume it is a sufficient statistic). If we assume discrete time steps, we can write the joint distribution as follows:

$$p(X_{1:T}) = p(X_1)p(X_2|X_1)p(X_3|X_2)\ldots = p(X_1)\prod_{t=2}^{T} p(X_t|X_{t-1}) \tag{17.1}$$

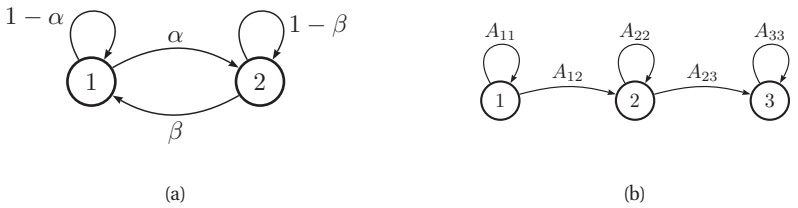
This is called a **Markov chain** or **Markov model**.

If we assume the transition function $p(X_t|X_{t-1})$ is independent of time, then the chain is called **homogeneous**, **stationary**, or **time-invariant**. This is an example of **parameter tying**, since the same parameter is shared by multiple variables. This assumption allows us to model an arbitrary number of variables using a fixed number of parameters; such models are called **stochastic processes**.

If we assume that the observed variables are discrete, so $X_t \in \{1, \ldots, K\}$, this is called a discrete-state or finite-state Markov chain. We will make this assumption throughout the rest of this section.

17.2.1 Transition matrix

When X_t is discrete, so $X_t \in \{1, \ldots, K\}$, the conditional distribution $p(X_t|X_{t-1})$ can be written as a $K \times K$ matrix, known as the **transition matrix A**, where $A_{ij} = p(X_t = j|X_{t-1} = i)$ is the probability of going from state i to state j. Each row of the matrix sums to one, $\sum_j A_{ij} = 1$, so this is called a **stochastic matrix**.



(a) (b)

Figure 17.1 State transition diagrams for some simple Markov chains. Left: a 2-state chain. Right: a 3-state left-to-right chain.

A stationary, finite-state Markov chain is equivalent to a **stochastic automaton**. It is common to visualize such automata by drawing a directed graph, where nodes represent states and arrows represent legal transitions, i.e., non-zero elements of $\mathbf{A}$. This is known as a **state transition diagram**. The weights associated with the arcs are the probabilities. For example, the following 2-state chain

$$\mathbf{A} = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{17.2}$$

is illustrated in Figure 17.1(left). The following 3-state chain

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & 0 \\ 0 & A_{22} & A_{23} \\ 0 & 0 & 1 \end{pmatrix} \tag{17.3}$$

is illustrated in Figure 17.1(right). This is called a **left-to-right transition matrix**, and is commonly used in speech recognition (Section 17.6.2).

The A_{ij} element of the transition matrix specifies the probability of getting from i to j in one step. The n-step transition matrix $\mathbf{A}(n)$ is defined as

$$A_{ij}(n) \triangleq p(X_{t+n} = j | X_t = i) \tag{17.4}$$

which is the probability of getting from i to j in exactly n steps. Obviously $\mathbf{A}(1) = \mathbf{A}$. The **Chapman-Kolmogorov** equations state that

$$A_{ij}(m+n) = \sum_{k=1}^{K} A_{ik}(m) A_{kj}(n) \tag{17.5}$$

In words, the probability of getting from i to j in $m+n$ steps is just the probability of getting from i to k in m steps, and then from k to j in n steps, summed up over all k. We can write the above as a matrix multiplication

$$\mathbf{A}(m+n) = \mathbf{A}(m)\mathbf{A}(n) \tag{17.6}$$

Hence

$$\mathbf{A}(n) = \mathbf{A}\ \mathbf{A}(n-1) = \mathbf{A}\ \mathbf{A}\ \mathbf{A}(n-2) = \cdots = \mathbf{A}^n \tag{17.7}$$

Thus we can simulate multiple steps of a Markov chain by "powering up" the transition matrix.

```
SAYS IT'S NOT IN THE CARDS LEGENDARY RECONNAISSANCE BY ROLLIE
DEMOCRACIES UNSUSTAINABLE COULD STRIKE REDLINING VISITS TO PROFIT
BOOKING WAIT HERE AT MADISON SQUARE GARDEN COUNTY COURTHOUSE WHERE HE
HAD BEEN DONE IN THREE ALREADY IN ANY WAY IN WHICH A TEACHER
```

Table 17.1 Example output from an 4-gram word model, trained using backoff smoothing on the Broadcast News corpus. The first 4 words are specified by hand, the model generates the 5th word, and then the results are fed back into the model. Source: `http://www.fit.vutbr.cz/~imikolov/rnnlm/gen-4gram.txt` .

17.2.2 Application: Language modeling

One important application of Markov models is to make statistical **language models**, which are probability distributions over sequences of words. We define the state space to be all the words in English (or some other language). The marginal probabilities $p(X_t = k)$ are called **unigram statistics**. If we use a first-order Markov model, then $p(X_t = k|X_{t-1} = j)$ is called a **bigram model**. If we use a second-order Markov model, then $p(X_t = k|X_{t-1} = j, X_{t-2} = i)$ is called a **trigram model**. And so on. In general these are called **n-gram models**. For example, Figure 17.2 shows 1-gram and 2-grams counts for the *letters* $\{a, \ldots, z, -\}$ (where - represents space) estimated from Darwin's *On The Origin Of Species*.

Language models can be used for several things, such as the following:

- **Sentence completion** A language model can predict the next word given the previous words in a sentence. This can be used to reduce the amount of typing required, which is particularly important for disabled users (see e.g., David Mackay's Dasher system[1]), or uses of mobile devices.
- **Data compression** Any density model can be used to define an encoding scheme, by assigning short codewords to more probable strings. The more accurate the predictive model, the fewer the number of bits it requires to store the data.
- **Text classification** Any density model can be used as a class-conditional density and hence turned into a (generative) classifier. Note that using a 0-gram class-conditional density (i.e., only unigram statistics) would be equivalent to a naive Bayes classifier (see Section 3.5).
- **Automatic essay writing** One can sample from $p(x_{1:t})$ to generate artificial text. This is one way of assessing the quality of the model. In Table 17.1, we give an example of text generated from a 4-gram model, trained on a corpus with 400 million words. ((Tomas et al. 2011) describes a much better language model, based on recurrent neural networks, which generates much more semantically plausible text.)

1. `http://www.inference.phy.cam.ac.uk/dasher/`

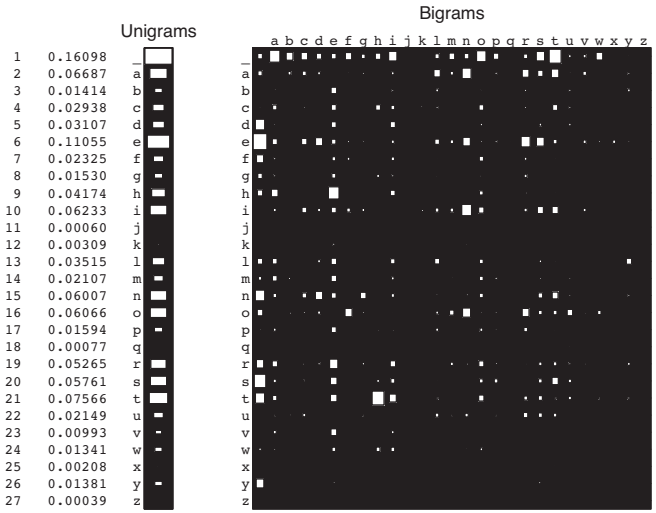


Figure 17.2 Unigram and bigram counts from Darwin's *On The Origin Of Species*. The 2D picture on the right is a Hinton diagram of the joint distribution. The size of the white squares is proportional to the value of the entry in the corresponding vector/ matrix. Based on (MacKay 2003, p22). Figure generated by `ngramPlot`.

17.2.2.1 MLE for Markov language models

We now discuss a simple way to estimate the transition matrix from training data. The probability of any particular sequence of length T is given by

$$p(x_{1:T}|\boldsymbol{\theta}) = \pi(x_1)A(x_1, x_2)\dots A(x_{T-1}, x_T) \tag{17.8}$$

$$= \prod_{j=1}^{K}(\pi_j)^{\mathbb{I}(x_1=j)} \prod_{t=2}^{T}\prod_{j=1}^{K}\prod_{k=1}^{K}(A_{jk})^{\mathbb{I}(x_t=k, x_{t-1}=j)} \tag{17.9}$$

Hence the log-likelihood of a set of sequences $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, where $\mathbf{x}_i = (x_{i1}, \dots, x_{i,T_i})$ is a sequence of length T_i, is given by

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{N}\log p(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_j N_j^1 \log \pi_j + \sum_j\sum_k N_{jk}\log A_{jk} \tag{17.10}$$

where we define the following counts:

$$N_j^1 \triangleq \sum_{i=1}^{N}\mathbb{I}(x_{i1}=j), \quad N_{jk} \triangleq \sum_{i=1}^{N}\sum_{t=1}^{T_i-1}\mathbb{I}(x_{i,t}=j, x_{i,t+1}=k) \tag{17.11}$$

Hence we can write the MLE as the normalized counts:

$$\hat{\pi}_j = \frac{N_j^1}{\sum_j N_j^1}, \quad \hat{A}_{jk} = \frac{N_{jk}}{\sum_k N_{jk}} \tag{17.12}$$

These results can be extended in a straightforward way to higher order Markov models. However, the problem of zero-counts becomes very acute whenever the number of states K, and/or the order of the chain, n, is large. An n-gram models has $O(K^n)$ parameters. If we have $K \sim 50,000$ words in our vocabulary, then a bi-gram model will have about 2.5 billion free parameters, corresponding to all possible word pairs. It is very unlikely we will see all of these in our training data. However, we do not want to predict that a particular word string is totally impossible just because we happen not to have seen it in our training text — that would be a severe form of overfitting.[2]

A simple solution to this is to use add-one smoothing, where we simply add one to all the empirical counts before normalizing. The Bayesian justification for this is given in Section 3.3.4.1. However add-one smoothing assumes all n-grams are equally likely, which is not very realistic. A more sophisticated Bayesian approach is discussed in Section 17.2.2.2.

An alternative to using smart priors is to gather lots and lots of data. For example, Google has fit n-gram models (for $n = 1 : 5$) based on one trillion words extracted from the web. Their data, which is over 100GB when uncompressed, is publically available.[3] An example of their data, for a set of 4-grams, is shown below.

```
serve as the incoming 92
serve as the incubator 99
serve as the independent 794
serve as the index 223
serve as the indication 72
serve as the indicator 120
serve as the indicators 45
serve as the indispensable 111
serve as the indispensible 40
serve as the individual 234
...
```

Although such an approach, based on "brute force and ignorance", can be successful, it is rather unsatisfying, since it is clear that this is not how humans learn (see e.g., (Tenenbaum and Xu 2000)). A more refined Bayesian approach, that needs much less data, is described in Section 17.2.2.2.

17.2.2.2 Empirical Bayes version of deleted interpolation

A common heuristic used to fix the sparse data problem is called **deleted interpolation** (Chen and Goodman 1996). This defines the transition matrix as a convex combination of the bigram

2. A famous example of an improbable, but syntactically valid, English word string, due to Noam Chomsky, is "colourless green ideas sleep furiously". We would not want our model to predict that this string is impossible. Even ungrammatical constructs should be allowed by our model with a certain probability, since people frequently violate grammatical rules, especially in spoken language.

3. See `http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html` for details.

frequencies $f_{jk} = N_{jk}/N_j$ and the unigram frequencies $f_k = N_k/N$:

$$A_{jk} = (1 - \lambda) f_{jk} + \lambda f_k \tag{17.13}$$

The term λ is usually set by cross validation. There is also a closely related technique called **backoff smoothing**; the idea is that if f_{jk} is too small, we "back off" to a more reliable estimate, namely f_k.

We will now show that the deleted interpolation heuristic is an approximation to the predictions made by a simple hierarchical Bayesian model. Our presentation follows (McKay and Peto 1995). First, let us use an independent Dirichlet prior on each row of the transition matrix:

$$\mathbf{A}_j \sim \text{Dir}(\alpha_0 m_1, \ldots, \alpha_0 m_K) = \text{Dir}(\alpha_0 \mathbf{m}) = \text{Dir}(\boldsymbol{\alpha}) \tag{17.14}$$

where $\mathbf{A}_j$ is row j of the transition matrix, $\mathbf{m}$ is the prior mean (satisfying $\sum_k m_k = 1$) and α_0 is the prior strength. We will use the same prior for each row: see Figure 17.3.

The posterior is given by $\mathbf{A}_j \sim \text{Dir}(\boldsymbol{\alpha} + \mathbf{N}_j)$, where $\mathbf{N}_j = (N_{j1}, \ldots, N_{jK})$ is the vector that records the number of times we have transitioned out of state j to each of the other states. From Equation 3.51, the posterior predictive density is

$$p(X_{t+1} = k | X_t = j, \mathcal{D}) = \overline{A}_{jk} = \frac{N_{jk} + \alpha m_k}{N_j + \alpha_0} = \frac{f_{jk} N_j + \alpha m_k}{N_j + \alpha_0} = (1 - \lambda_j) f_{jk} + \lambda_j m_k \tag{17.15}$$

where $\overline{A}_{jk} = \mathbb{E}[A_{jk} | \mathcal{D}, \boldsymbol{\alpha}]$ and

$$\lambda_j = \frac{\alpha}{N_j + \alpha_0} \tag{17.16}$$

This is very similar to Equation 17.13 but not identical. The main difference is that the Bayesian model uses a context-dependent weight λ_j to combine m_k with the empirical frequency f_{jk}, rather than a fixed weight λ. This is like *adaptive* deleted interpolation. Furthermore, rather than backing off to the empirical marginal frequencies f_k, we back off to the model parameter m_k.

The only remaining question is: what values should we use for α and $\mathbf{m}$? Let's use empirical Bayes. Since we assume each row of the transition matrix is a priori independent given $\boldsymbol{\alpha}$, the marginal likelihood for our Markov model is found by applying Equation 5.24 to each row:

$$p(\mathcal{D} | \boldsymbol{\alpha}) = \prod_j \frac{B(\mathbf{N}_j + \boldsymbol{\alpha})}{B(\boldsymbol{\alpha})} \tag{17.17}$$

where $\mathbf{N}_j = (N_{j1}, \ldots, N_{jK})$ are the counts for leaving state j and $B(\boldsymbol{\alpha})$ is the generalized beta function.

We can fit this using the methods discussed in (Minka 2000e). However, we can also use the following approximation (McKay and Peto 1995, p12):

$$m_k \propto |\{j : N_{jk} > 0\}| \tag{17.18}$$

This says that the prior probability of word k is given by the number of different contexts in which it occurs, rather than the number of times it occurs. To justify the reasonableness of this result, Mackay and Peto (McKay and Peto 1995) give the following example.

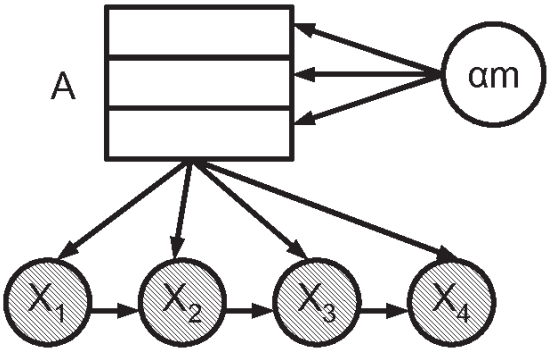


Figure 17.3 A Markov chain in which we put a different Dirichlet prior on every row of the transition matrix **A**, but the hyperparameters of the Dirichlet are shared.

```
Imagine, you see, that the language, you see, has, you see, a
frequently occuring couplet 'you see', you see, in which the second
word of the couplet, see, follows the first word, you, with very high
probability, you see. Then the marginal statistics, you see, are going
to become hugely dominated, you see, by the words you and see, with
equal frequency, you see.
```

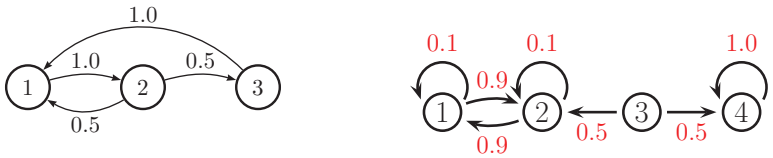
If we use the standard smoothing formula, Equation 17.13, then P(you|novel) and P(see|novel), for some novel context word not seen before, would turn out to be the same, since the marginal frequencies of 'you' and 'see' are the same (11 times each). However, this seems unreasonable. 'You' appears in many contexts, so P(you|novel) should be high, but 'see' only follows 'you', so P(see|novel) should be low. If we use the Bayesian formula Equation 17.15, we will get this effect for free, since we back off to m_k not f_k, and m_k will be large for 'you' and small for 'see' by Equation 17.18.

Unfortunately, although elegant, this Bayesian model does not beat the state-of-the-art language model, known as **interpolated Kneser-Ney** (Kneser and Ney 1995; Chen and Goodman 1998). However, in (Teh 2006), it was shown how one can build a non-parametric Bayesian model which outperforms interpolated Kneser-Ney, by using variable-length contexts. In (Wood et al. 2009), this method was extended to create the "sequence memoizer", which is currently (2010) the best-performing language model.[4]

17.2.2.3 Handling out-of-vocabulary words

While the above smoothing methods handle the case where the counts are small or even zero, none of them deal with the case where the test set may contain a completely novel word. In particular, they all assume that the words in the vocabulary (i.e., the state space of X_t) is fixed and known (typically it is the set of unique words in the training data, or in some dictionary).

4. Interestingly, these non-parametric methods are based on posterior inference using MCMC (Section 24.1) and/or particle filtering (Section 23.5), rather than optimization methods such as EB. Despite this, they are quite efficient.



(a) (b)

Figure 17.4 Some Markov chains. (a) A 3-state aperiodic chain. (b) A reducible 4-state chain.

Even if all $\overline{A}_{jk}$'s are non-zero, none of these models will predict a novel word outside of this set, and hence will assign zero probability to a test sentence with an unfamiliar word. (Unfamiliar words are bound to occur, because the set of words is an open class. For example, the set of proper nouns (names of people and places) is unbounded.)

A standard heuristic to solve this problem is to replace all novel words with the special symbol **unk**, which stands for "unknown". A certain amount of probability mass is held aside for this event.

A more principled solution would be to use a Dirichlet process, which can generate a countably infinite state space, as the amount of data increases (see Section 25.2.2). If all novel words are "accepted" as genuine words, then the system has no predictive power, since any misspelling will be considered a new word. So the novel word has to be seen frequently enough to warrant being added to the vocabulary. See e.g., (Friedman and Singer 1999; Griffiths and Tenenbaum 2001) for details.

17.2.3 Stationary distribution of a Markov chain *

We have been focussing on Markov models as a way of defining joint probability distributions over sequences. However, we can also interpret them as stochastic dynamical systems, where we "hop" from one state to another at each time step. In this case, we are often interested in the long term distribution over states, which is known as the **stationary distribution** of the chain. In this section, we discuss some of the relevant theory. Later we will consider two important applications: Google's PageRank algorithm for ranking web pages (Section 17.2.4), and the MCMC algorithm for generating samples from hard-to-normalize probability distributions (Chapter 24).

17.2.3.1 What is a stationary distribution?

Let $A_{ij} = p(X_t = j|X_{t-1} = i)$ be the one-step transition matrix, and let $\pi_t(j) = p(X_t = j)$ be the probability of being in state j at time t. It is conventional in this context to assume that $\boldsymbol{\pi}$ is a *row* vector. If we have an initial distribution over states of $\boldsymbol{\pi}_0$, then at time 1 we have

$$\pi_1(j) = \sum_i \pi_0(i) A_{ij} \tag{17.19}$$

or, in matrix notation,

$$\boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 \mathbf{A} \tag{17.20}$$

We can imagine iterating these equations. If we ever reach a stage where

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A} \tag{17.21}$$

then we say we have reached the **stationary distribution** (also called the **invariant distribution** or **equilibrium distribution**). Once we enter the stationary distribution, we will never leave.

For example, consider the chain in Figure 17.4(a). To find its stationary distribution, we write

$$\begin{pmatrix}\pi_1 & \pi_2 & \pi_3\end{pmatrix} = \begin{pmatrix}\pi_1 & \pi_2 & \pi_3\end{pmatrix} \begin{pmatrix} 1-A_{12}-A_{13} & A_{12} & A_{13} \\ A_{21} & 1-A_{21}-A_{23} & A_{23} \\ A_{31} & A_{32} & 1-A_{31}-A_{32} \end{pmatrix} \tag{17.22}$$

so

$$\pi_1 = \pi_1(1 - A_{12} - A_{12}) + \pi_2 A_{21} + \pi_3 A_{31} \tag{17.23}$$

or

$$\pi_1(A_{12} + A_{13}) = \pi_2 A_{21} + \pi_3 A_{31} \tag{17.24}$$

In general, we have

$$\pi_i \sum_{j \neq i} A_{ij} = \sum_{j \neq i} \pi_j A_{ji} \tag{17.25}$$

In other words, the probability of being in state i times the net flow out of state i must equal the probability of being in each other state j times the net flow from that state into i. These are called the **global balance equations**. We can then solve these equations, subject to the constraint that $\sum_j \pi_j = 1$.

17.2.3.2 Computing the stationary distribution

To find the stationary distribution, we can just solve the eigenvector equation $\mathbf{A}^T\mathbf{v} = \mathbf{v}$, and then to set $\boldsymbol{\pi} = \mathbf{v}^T$, where $\mathbf{v}$ is an eigenvector with eigenvalue 1. (We can be sure such an eigenvector exists, since $\mathbf{A}$ is a row-stochastic matrix, so $\mathbf{A1} = \mathbf{1}$; also recall that the eigenvalues of $\mathbf{A}$ and $\mathbf{A}^T$ are the same.) Of course, since eigenvectors are unique only up to constants of proportionality, we must normalize $\mathbf{v}$ at the end to ensure it sums to one.

Note, however, that the eigenvectors are only guaranteed to be real-valued if the matrix is positive, $A_{ij} > 0$ (and hence $A_{ij} < 1$, due to the sum-to-one constraint). A more general approach, which can handle chains where some transition probabilities are 0 or 1 (such as Figure 17.4(a)), is as follows (Resnick 1992, p138). We have K constraints from $\boldsymbol{\pi}(\mathbf{I} - \mathbf{A}) = \mathbf{0}_{K\times 1}$ and 1 constraint from $\boldsymbol{\pi}\mathbf{1}_{K\times 1} = 0$. Since we only have K unknowns, this is overconstrained. So let us replace any column (e.g., the last) of $\mathbf{I} - \mathbf{A}$ with $\mathbf{1}$, to get a new matrix, call it $\mathbf{M}$. Next we define $\mathbf{r} = [0, 0, \ldots, 1]$, where the 1 in the last position corresponds to the column of all 1s in $\mathbf{M}$. We then solve $\boldsymbol{\pi}\mathbf{M} = \mathbf{r}$. For example, for a 3 state chain we have to solve this linear system:

$$\begin{pmatrix}\pi_1 & \pi_2 & \pi_3\end{pmatrix} \begin{pmatrix} 1-A_{11} & -A_{12} & 1 \\ -A_{21} & 1-A_{22} & 1 \\ -A_{31} & -A_{32} & 1 \end{pmatrix} = \begin{pmatrix}0 & 0 & 1\end{pmatrix} \tag{17.26}$$

For the chain in Figure 17.4(a) we find $\boldsymbol{\pi} = [0.4, 0.4, 0.2]$. We can easily verify this is correct, since $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A}$. See `mcStatDist` for some Matlab code.

Unfortunately, not all chains have a stationary distribution. as we explain below.

17.2.3.3 When does a stationary distribution exist? *

Consider the 4-state chain in Figure 17.4(b). If we start in state 4, we will stay there forever, since 4 is an **absorbing state**. Thus $\boldsymbol{\pi} = (0, 0, 0, 1)$ is one possible stationary distribution. However, if we start in 1 or 2, we will oscillate between those two states for ever. So $\boldsymbol{\pi} = (0.5, 0.5, 0, 0)$ is another possible stationary distribution. If we start in state 3, we could end up in either of the above stationary distributions.

We see from this example that a necessary condition to have a unique stationary distribution is that the state transition diagram be a singly connected component, i.e., we can get from any state to any other state. Such chains are called **irreducible**.

Now consider the 2-state chain in Figure 17.1(a). This is irreducible provided $\alpha, \beta > 0$. Suppose $\alpha = \beta = 0.9$. It is clear by symmetry that this chain will spend 50% of its time in each state. Thus $\boldsymbol{\pi} = (0.5, 0.5)$. But now suppose $\alpha = \beta = 1$. In this case, the chain will oscillate between the two states, but the long-term distribution on states depends on where you start from. If we start in state 1, then on every odd time step (1,3,5,...) we will be in state 1; but if we start in state 2, then on every odd time step we will be in state 2.

This example motivates the following definition. Let us say that a chain has a **limiting distribution** if $\pi_j = \lim_{n\to\infty} A_{ij}^n$ exists and is independent of i, for all j. If this holds, then the long-run distribution over states will be independent of the starting state:

$$P(X_t = j) = \sum_i P(X_0 = i) A_{ij}(t) \to \pi_j \text{ as } t \to \infty \tag{17.27}$$

Let us now characterize when a limiting distribution exists. Define the **period** of state i to be

$$d(i) = \text{gcd}\{t : A_{ii}(t) > 0\} \tag{17.28}$$

where gcd stands for **greatest common divisor**, i.e., the largest integer that divides all the members of the set. For example, in Figure 17.4(a), we have $d(1) = d(2) = gcd(2, 3, 4, 6, ...) = 1$ and $d(3) = gcd(3, 5, 6, ...) = 1$. We say a state i is **aperiodic** if $d(i) = 1$. (A sufficient condition to ensure this is if state i has a self-loop, but this is not a necessary condition.) We say a chain is aperiodic if all its states are aperiodic. One can show the following important result:

Theorem 17.2.1. *Every irreducible (singly connected), aperiodic finite state Markov chain has a limiting distribution, which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

A special case of this result says that every regular finite state chain has a unique stationary distribution, where a **regular** chain is one whose transition matrix satisfies $A_{ij}^n > 0$ for some integer n and all i, j, i.e., it is possible to get from any state to any other state in n steps. Consequently, after n steps, the chain could be in any state, no matter where it started. One can show that sufficient conditions to ensure regularity are that the chain be irreducible (singly connected) and that every state have a self-transition.

To handle the case of Markov chains whose state-space is not finite (e.g, the countable set of all integers, or all the uncountable set of all reals), we need to generalize some of the earlier

definitions. Since the details are rather technical, we just briefly state the main results without proof. See e.g., (Grimmett and Stirzaker 1992) for details.

For a stationary distribution to exist, we require irreducibility (singly connected) and aperiodicity, as before. But we also require that each state is **recurrent**. (A chain in which all states are recurrent is called a recurrent chain.) Recurrent means that you will return to that state with probability 1. As a simple example of a non-recurrent state (i.e., a **transient** state), consider Figure 17.4(b): states 3 is transient because one immediately leaves it and either spins around state 4 forever, or oscillates between states 1 and 2 forever. There is no way to return to state 3.

It is clear that any finite-state irreducible chain is recurrent, since you can always get back to where you started from. But now consider an example with an infinite state space. Suppose we perform a random walk on the integers, $\mathcal{X} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Let $A_{i,i+1} = p$ be the probability of moving right, and $A_{i,i-1} = 1 - p$ be the probability of moving left. Suppose we start at $X_1 = 0$. If $p > 0.5$, we will shoot off to $+\infty$; we are not guaranteed to return. Similarly, if $p < 0.5$, we will shoot off to $-\infty$. So in both cases, the chain is not recurrent, even though it is irreducible.

It should be intuitively obvious that we require all states to be recurrent for a stationary distribution to exist. However, this is not sufficient. To see this, consider the random walk on the integers again, and suppose $p = 0.5$. In this case, we can return to the origin an infinite number of times, so the chain is recurrent. However, it takes infinitely long to do so. This prohibits it from having a stationary distribution. The intuitive reason is that the distribution keeps spreading out over a larger and larger set of the integers, and never converges to a stationary distribution. More formally, we define a state to be **non-null recurrent** if the expected time to return to this state is finite. A chain in which all states are non-null is called a non-null chain.

For brevity, we we say that a state is **ergodic** if it is aperiodic, recurrent and non-null, and we say a chain is ergodic if all its states are ergodic.

We can now state our main theorem:

Theorem 17.2.2. *Every irreducible (singly connected), ergodic Markov chain has a limiting distribution, which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

This generalizes Theorem 17.2.1, since for irreducible finite-state chains, all states are recurrent and non-null.

17.2.3.4 Detailed balance

Establishing ergodicity can be difficult. We now give an alternative condition that is easier to verify.

We say that a Markov chain $\mathbf{A}$ is **time reversible** if there exists a distribution $\boldsymbol{\pi}$ such that

$$\pi_i A_{ij} = \pi_j A_{ji} \tag{17.29}$$

These are called the **detailed balance equations**. This says that the flow from i to j must equal the flow from j to i, weighted by the appropriate source probabilities.

We have the following important result.

Theorem 17.2.3. *If a Markov chain with transition matrix $\mathbf{A}$ is regular and satisfies detailed balance wrt distribution $\boldsymbol{\pi}$, then $\boldsymbol{\pi}$ is a stationary distribution of the chain.*

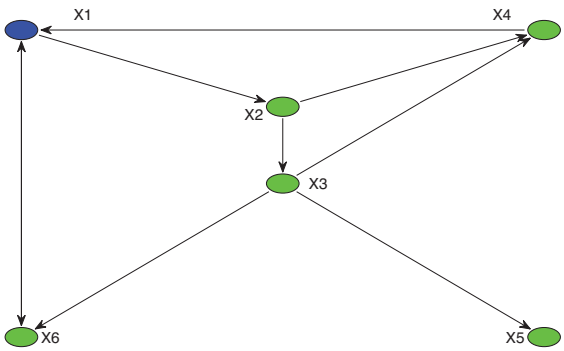


Figure 17.5 A very small world wide web. Figure generated by `pagerankDemo`, written by Tim Davis.

Proof. To see this, note that

$$\sum_i \pi_i A_{ij} = \sum_i \pi_j A_{ji} = \pi_j \sum_i A_{ji} = \pi_j \tag{17.30}$$

and hence $\boldsymbol{\pi} = \mathbf{A}\boldsymbol{\pi}$. □

Note that this condition is sufficient but not necessary (see Figure 17.4(a) for an example of a chain with a stationary distribution which does not satisfy detailed balance).

In Section 24.1, we will discuss Markov chain Monte Carlo or MCMC methods. These take as input a desired distribution $\boldsymbol{\pi}$ and construct a transition matrix (or in general, a transition **kernel**) $\mathbf{A}$ which satisfies detailed balance wrt $\boldsymbol{\pi}$. Thus by sampling states from such a chain, we will eventually enter the stationary distribution, and will visit states with probabilities given by $\boldsymbol{\pi}$.

17.2.4 Application: Google's PageRank algorithm for web page ranking *

The results in Section 17.2.3 form the theoretical underpinnings to Google's **PageRank** algorithm, which is used for information retrieval on the world-wide web. We sketch the basic idea below; see (Byran and Leise 2006) for a more detailed explanation.

We will treat the web as a giant directed graph, where nodes represent web pages (documents) and edges represent hyper-links.[5] We then perform a process called **web crawling**. We start at a few designated root nodes, such as `dmoz.org`, the home of the Open Directory Project, and then follows the links, storing all the pages that we encounter, until we run out of time.

Next, all of the words in each web page are entered into a data structure called an **inverted index**. That is, for each word, we store a list of the documents where this word occurs. (In practice, we store a list of hash codes representing the URLs.) At test time, when a user enters

5. In 2008, Google said it had indexed 1 trillion (10^{12}) unique URLs. If we assume there are about 10 URLs per page (on average), this means there were about 100 billion unique web pages. Estimates for 2010 are about 121 billion unique web pages. Source: `thenextweb.com/shareables/2011/01/11/infographic-how-big-is-the-internet`.

a query, we can just look up all the documents containing each word, and intersect these lists (since queries are defined by a conjunction of search terms). We can get a refined search by storing the location of each word in each document. We can then test if the words in a document occur in the same order as in the query.

Let us give an example, from `http://en.wikipedia.org/wiki/Inverted_index`. We have 3 documents, T_0 = "it is what it is", T_1 = "what is it" and T_2 = "it is a banana". Then we can create the following inverted index, where each pair represents a document and word location:

```
"a":      {(2, 2)}
"banana": {(2, 3)}
"is":     {(0, 1), (0, 4), (1, 1), (2, 1)}
"it":     {(0, 0), (0, 3), (1, 2), (2, 0)}
"what":   {(0, 2), (1, 0)}
```

For example, we see that the word "what" occurs at location 2 (counting from 0) in document 0, and location 0 in document 1. Suppose we search for "what is it". If we ignore word order, we retrieve the following documents:

$$\{T_0, T_1\} \cap \{T_0, T_1, T_2\} \cap \{T_0, T_1, T_2\} = \{T_0, T_1\} \tag{17.31}$$

If we require that the word order matches, only document T_1 would be returned. More generally, we can allow out-of-order matches, but can give "bonus points" to documents whose word order matches the query's word order, or to other features, such as if the words occur in the title of a document. We can then return the matching documents in decreasing order of their score/relevance. This is called document **ranking**.

So far, we have described the standard process of information retrieval. But the link structure of the web provides an additional source of information. The basic idea is that some web pages are more authoritative than others, so these should be ranked higher (assuming they match the query). A web page is an authority if it is linked to by many other pages. But to protect against the effect of so-called **link farms**, which are dummy pages which just link to a given site to boost its apparent relevance, we will weight each incoming link by the source's authority. Thus we get the following recursive definition for the authoritativeness of page j, also called its **PageRank**:

$$\pi_j = \sum_i A_{ij} \pi_i \tag{17.32}$$

where A_{ij} is the probability of following a link from i to j. We recognize Equation 17.32 as the stationary distribution of a Markov chain.

In the simplest setting, we define $A_{i.}$ as a uniform distribution over all states that i is connected to. However, to ensure the distribution is unique, we need to make the chain into a regular chain. This can be done by allowing each state i to jump to any other state (including itself) with some small probability. This effectively makes the transition matrix aperiodic and fully connected (although the adjacency matrix G_{ij} of the web itself is highly sparse).

We discuss efficient methods for computing the leading eigenvector of this giant matrix below. But first, let us give an example of the PageRank algorithm. Consider the small web in Figure 17.5.

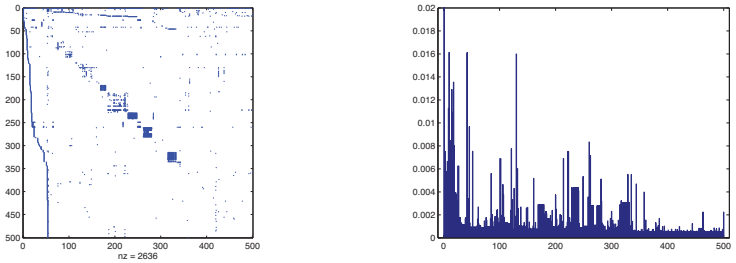


Figure 17.6 (a) Web graph of 500 sites rooted at `www.harvard.edu`. (b) Corresponding page rank vector. Figure generated by `pagerankDemoPmtk`, Based on code by Cleve Moler (Moler 2004).

We find that the stationary distribution is

$$\boldsymbol{\pi} = (0.3209, 0.1706, 0.1065, 0.1368, 0.0643, 0.2008) \tag{17.33}$$

So a random surfer will visit site 1 about 32% of the time. We see that node 1 has a higher PageRank than nodes 4 or 6, even though they all have the same number of in-links. This is because being linked to from an influential nodehelps increase your PageRank score more than being linked to by a less influential node.

As a slightly larger example, Figure 17.6(a) shows a web graph, derived from the root of `harvard.edu`. Figure 17.6(b) shows the corresponding PageRank vector.

17.2.4.1 Efficiently computing the PageRank vector

Let $G_{ij} = 1$ iff there is a link from j to i. Now imagine performing a random walk on this graph, where at every time step, with probability $p = 0.85$ you follow one of the outlinks uniformly at random, and with probability $1 - p$ you jump to a random node, again chosen uniformly at random. If there are no outlinks, you just jump to a random page. (These random jumps, including self-transitions, ensure the chain is irreducible (singly connected) and regular. Hence we can solve for its unique stationary distribution using eigenvector methods.) This defines the following transition matrix:

$$M_{ij} = \begin{cases} pG_{ij}/c_j + \delta & \text{if } c_j \neq 0 \\ 1/n & \text{if } c_j = 0 \end{cases} \tag{17.34}$$

where n is the number of nodes, $\delta = (1-p)/n$ is the probability of jumping from one page to another without following a link and $c_j = \sum_i G_{ij}$ represents the out-degree of page j. (If $n = 4 \cdot 10^9$ and $p = 0.85$, then $\delta = 3.75 \cdot 10^{-11}$.) Here $\mathbf{M}$ is a stochastic matrix in which *columns* sum to one. Note that $\mathbf{M} = \mathbf{A}^T$ in our earlier notation.

We can represent the transition matrix compactly as follows. Define the diagonal matrix $\mathbf{D}$ with entries

$$d_{jj} = \begin{cases} 1/c_j & \text{if } c_j \neq 0 \\ 0 & \text{if } c_j = 0 \end{cases} \tag{17.35}$$

Define the vector $\mathbf{z}$ with components

$$z_j = \begin{cases} \delta & \text{if } c_j \neq 0 \\ 1/n & \text{if } c_j = 0 \end{cases} \tag{17.36}$$

Then we can rewrite Equation 17.34 as follows:

$$\mathbf{M} = p\mathbf{G}\mathbf{D} + \mathbf{1}\mathbf{z}^T \tag{17.37}$$

The matrix $\mathbf{M}$ is not sparse, but it is a rank one modification of a sparse matrix. Most of the elements of $\mathbf{M}$ are equal to the small constant δ. Obviously these do not need to be stored explicitly.

Our goal is to solve $\mathbf{v} = \mathbf{M}\mathbf{v}$, where $\mathbf{v} = \boldsymbol{\pi}^T$. One efficient method to find the leading eigenvector of a large matrix is known as the **power method**. This simply consists of repeated matrix-vector multiplication, followed by normalization:

$$\mathbf{v} \propto \mathbf{M}\mathbf{v} = p\mathbf{G}\mathbf{D}\mathbf{v} + \mathbf{1}\mathbf{z}^T\mathbf{v} \tag{17.38}$$

It is possible to implement the power method without using any matrix multiplications, by simply sampling from the transition matrix and counting how often you visit each state. This is essentially a Monte Carlo approximation to the sum implied by $\mathbf{v} = \mathbf{M}\mathbf{v}$. Applying this to the data in Figure 17.6(a) yields the stationary distribution in Figure 17.6(b). This took 13 iterations to converge, starting from a uniform distribution. (See also the function `pagerankDemo`, by Tim Davis, for an animation of the algorithm in action, applied to the small web example.) To handle changing web structure, we can re-run this algorithm every day or every week, starting $\mathbf{v}$ off at the old distribution (Langville and Meyer 2006).

For details on how to perform this Monte Carlo power method in a parallel distributed computing environment, see e.g., (Rajaraman and Ullman 2010).

17.2.4.2 Web spam

PageRank is not foolproof. For example, consider the strategy adopted by JC Penney, a department store in the USA. During the Christmas season of 2010, it planted many links to its home page on 1000s of irrelevant web pages, thus increasing its ranking on Google's search engine (Segal 2011). Even though each of these source pages has low PageRank, there were so many of them that their effect added up. Businesses call this **search engine optimization**; Google calls it **web spam**. When Google was notified of this scam (by the *New York Times*), it manually downweighted JC Penney, since such behavior violates Google's code of conduct. The result was that JC Penney dropped from rank 1 to rank 65, essentially making it disappear from view. Automatically detecting such scams relies on various techniques which are beyond the scope of this chapter.

17.3 Hidden Markov models

As we mentioned in Section 10.2.2, a **hidden Markov model** or **HMM** consists of a discrete-time, discrete-state Markov chain, with hidden states $z_t \in \{1, \ldots, K\}$, plus an **observation** model

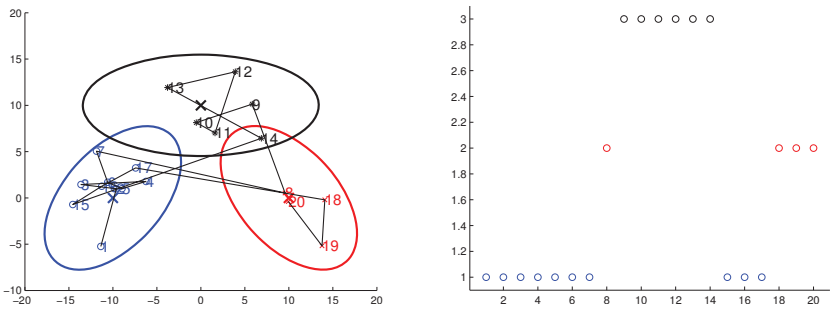


Figure 17.7 (a) Some 2d data sampled from a 3 state HMM. Each state emits from a 2d Gaussian. (b) The hidden state sequence. Based on Figure 13.8 of (Bishop 2006b). Figure generated by `hmmLillypadDemo`.

$p(\mathbf{x}_t|z_t)$. The corresponding joint distribution has the form

$$p(\mathbf{z}_{1:T}, \mathbf{x}_{1:T}) = p(\mathbf{z}_{1:T})p(\mathbf{x}_{1:T}|\mathbf{z}_{1:T}) = \left[p(z_1)\prod_{t=2}^{T} p(z_t|z_{t-1})\right]\left[\prod_{t=1}^{T} p(\mathbf{x}_t|z_t)\right] \tag{17.39}$$

The observations in an HMM can be discrete or continuous. If they are discrete, it is common for the observation model to be an observation matrix:

$$p(\mathbf{x}_t = l|z_t = k, \boldsymbol{\theta}) = B(k, l) \tag{17.40}$$

If the observations are continuous, it is common for the observation model to be a conditional Gaussian:

$$p(\mathbf{x}_t|z_t = k, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{17.41}$$

Figure 17.7 shows an example where we have 3 states, each of which emits a different Gaussian. The resulting model is similar to a Gaussian mixture model, except the cluster membership has Markovian dynamics. (Indeed, HMMs are sometimes called **Markov switching models** (Fruhwirth-Schnatter 2007).) We see that we tend to get multiple observations in the same location, and then a sudden jump to a new cluster.

17.3.1 Applications of HMMs

HMMs can be used as black-box density models on sequences. They have the advantage over Markov models in that they can represent long-range dependencies between observations, mediated via the latent variables. In particular, note that they do not assume the Markov property holds for the observations themselves. Such black-box models are useful for time-series prediction (Fraser 2008). They can also be used to define class-conditional densities inside a generative classifier.

However, it is more common to imbue the hidden states with some desired meaning, and to then try to estimate the hidden states from the observations, i.e., to compute $p(z_t|\mathbf{x}_{1:t})$ if we are

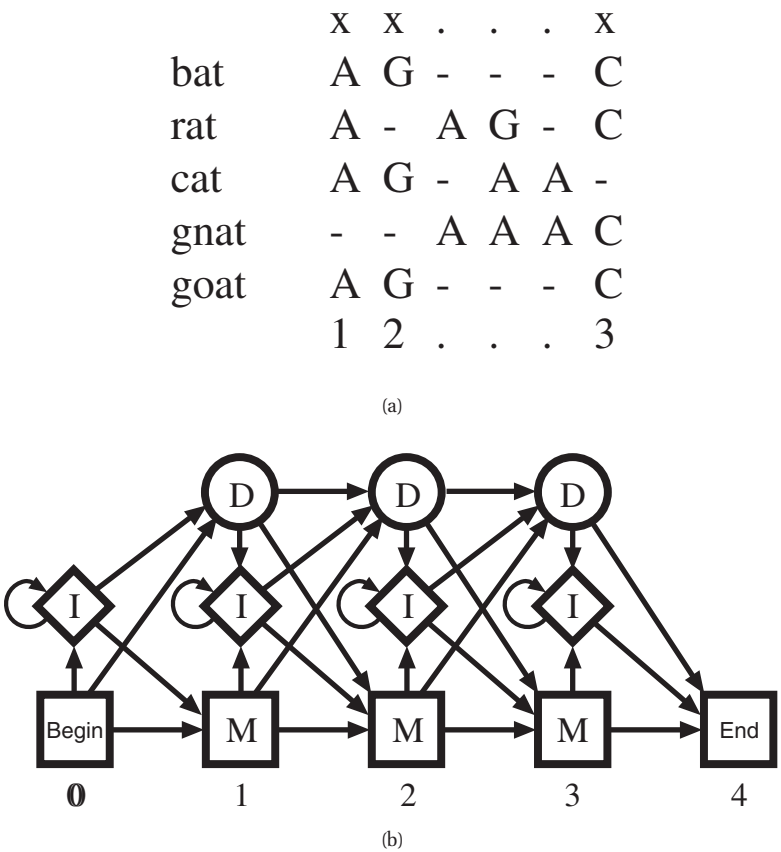


Figure 17.8 (a) Some DNA sequences. (b) State transition diagram for a profile HMM. Source: Figure 5.7 of (Durbin et al. 1998). Used with kind permission of Richard Durbin.

in an online scenario, or $p(z_t|\mathbf{x}_{1:T})$ if we are in an offline scenario (see Section 17.4.1 for further discussion of the differences between these two approaches). Below we give some examples of applications which use HMMs in this way:

- **Automatic speech recognition**. Here $\mathbf{x}_t$ represents features extracted from the speech signal, and z_t represents the word that is being spoken. The transition model $p(z_t|z_{t-1})$ represents the language model, and the observation model $p(\mathbf{x}_t|z_t)$ represents the acoustic model. See e.g., (Jelinek 1997; Jurafsky and Martin 2008) for details.
- **Activity recognition**. Here $\mathbf{x}_t$ represents features extracted from a video frame, and z_t is the class of activity the person is engaged in (e.g., running, walking, sitting, etc.) See e.g., (Szeliski 2010) for details.
- **Part of speech tagging**. Here x_t represents a word, and z_t represents its **part of speech** (noun, verb, adjective, etc.) See Section 19.6.2.1 for more information on POS tagging and

related tasks.

- **Gene finding**. Here x_t represents the DNA nucleotides (A,C,G,T), and z_t represents whether we are inside a gene-coding region or not. See e.g., (Schweikerta et al. 2009) for details.
- **Protein sequence alignment**. Here x_t represents an amino acid, and z_t represents whether this matches the latent **consensus sequence** at this location. This model is called a **profile HMM** and is illustrated in Figure 17.8. The HMM has 3 states, called match, insert and delete. If z_t is a match state, then x_t is equal to the t'th value of the consensus. If z_t is an insert state, then x_t is generated from a uniform distribution that is unrelated to the consensus sequence. If z_t is a delete state, then $x_t = -$. In this way, we can generate noisy copies of the consensus sequence of different lengths. In Figure 17.8(a), the consensus is "AGC", and we see various versions of this below. A path through the state transition diagram, shown in Figure 17.8(b), specifies how to align a sequence to the consensus, e.g., for the gnat, the most probable path is D, D, I, I, I, M. This means we delete the A and G parts of the consensus sequence, we insert 3 A's, and then we match the final C. We can estimate the model parameters by counting the number of such transitions, and the number of emissions from each kind of state, as shown in Figure 17.8(c). See Section 17.5 for more information on training an HMM, and (Durbin et al. 1998) for details on profile HMMs.

Note that for some of these tasks, conditional random fields, which are essentially discriminative versions of HMMs, may be more suitable; see Chapter 19 for details.

17.4 Inference in HMMs

We now discuss how to infer the hidden state sequence of an HMM, assuming the parameters are known. Exactly the same algorithms apply to other chain-structured graphical models, such as chain CRFs (see Section 19.6.1). In Chapter 20, we generalize these methods to arbitrary graphs. And in Section 17.5.2, we show how we can use the output of inference in the context of parameter estimation.

17.4.1 Types of inference problems for temporal models

There are several different kinds of inferential tasks for an HMM (and SSM in general). To illustrate the differences, we will consider an example called the **occasionally dishonest casino**, from (Durbin et al. 1998). In this model, $x_t \in \{1, 2, \ldots, 6\}$ represents which dice face shows up, and z_t represents the identity of the dice that is being used. Most of the time the casino uses a fair dice, $z = 1$, but occasionally it switches to a loaded dice, $z = 2$, for a short period. If $z = 1$ the observation distribution is a uniform multinoulli over the symbols $\{1, \ldots, 6\}$. If $z = 2$, the observation distribution is skewed towards face 6 (see Figure 17.9). If we sample from this model, we may observe data such as the following:

Listing 17.1 Example output of `casinoDemo`

```
Rolls:    664153216162115234653214356634261655234232315142464156663246
Die:      LLLLLLLLLLLLLLFFFFFFLLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFLLLLLLL
```

Here "rolls" refers to the observed symbol and "die" refers to the hidden state (L is loaded and F is fair). Thus we see that the model generates a sequence of symbols, but the statistics of the

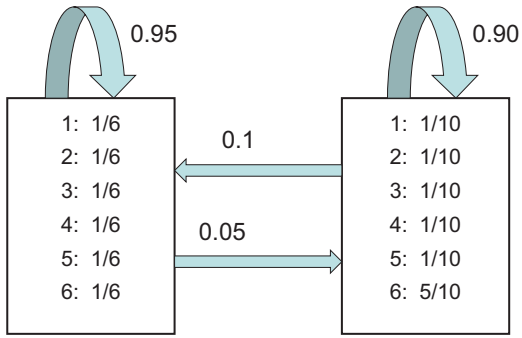
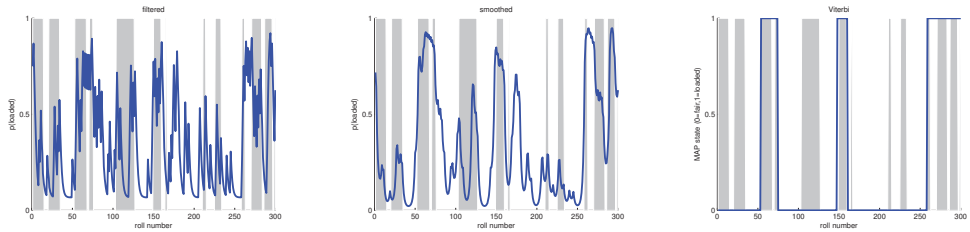


Figure 17.9 An HMM for the occasionally dishonest casino. The blue arrows visualize the state transition diagram $\mathbf{A}$. Based on (Durbin et al. 1998, p54).



(a) (b) (c)

Figure 17.10 Inference in the dishonest casino. Vertical gray bars denote the samples that we generated using a loaded die. (a) Filtered estimate of probability of using a loaded dice. (b) Smoothed estimates. (c) MAP trajectory. Figure generated by `casinoDemo`.

distribution changes abruptly every now and then. In a typical application, we just see the rolls and want to infer which dice is being used. But there are different kinds of inference, which we summarize below.

- **Filtering** means to compute the **belief state** $p(z_t|\mathbf{x}_{1:t})$ online, or recursively, as the data streams in. This is called "filtering" because it reduces the noise more than simply estimating the hidden state using just the current estimate, $p(z_t|\mathbf{x}_t)$. We will see below that we can perform filtering by simply applying Bayes rule in a sequential fashion. See Figure 17.10(a) for an example.
- **Smoothing** means to compute $p(z_t|\mathbf{x}_{1:T})$ offline, given all the evidence. See Figure 17.10(b) for an example. By conditioning on past and future data, our uncertainty will be significantly reduced. To understand this intuitively, consider a detective trying to figure out who committed a crime. As he moves through the crime scene, his uncertainty is high until he finds the key clue; then he has an "aha" moment, his uncertainty is reduced, and all the previously confusing observations are, in **hindsight**, easy to explain.

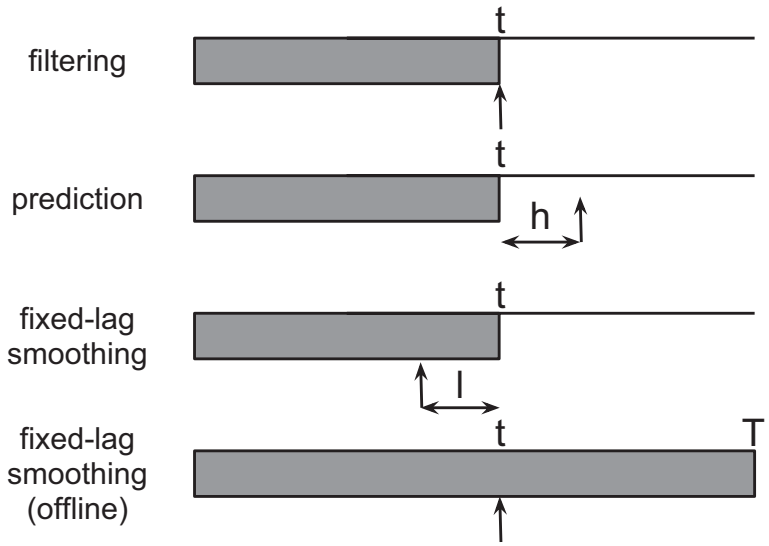


Figure 17.11 The main kinds of inference for state-space models. The shaded region is the interval for which we have data. The arrow represents the time step at which we want to perform inference. t is the current time, T is the sequence length, ℓ is the lag and h is the prediction horizon. See text for details.

- **Fixed lag smoothing** is an interesting compromise between online and offline estimation; it involves computing $p(z_{t-\ell}|\mathbf{x}_{1:t})$, where $\ell > 0$ is called the lag. This gives better performance than filtering, but incurs a slight delay. By changing the size of the lag, one can trade off accuracy vs delay.
- **Prediction** Instead of predicting the past given the future, as in fixed lag smoothing, we might want to predict the future given the past, i.e., to compute $p(z_{t+h}|\mathbf{x}_{1:t})$, where $h > 0$ is called the prediction **horizon**. For example, suppose $h = 2$; then we have

$$p(z_{t+2}|\mathbf{x}_{1:t}) = \sum_{z_{t+1}} \sum_{z_t} p(z_{t+2}|z_{t+1}) p(z_{t+1}|z_t) p(z_t|\mathbf{x}_{1:t}) \tag{17.42}$$

  It is straightforward to perform this computation: we just power up the transition matrix and apply it to the current belief state. The quantity $p(z_{t+h}|\mathbf{x}_{1:t})$ is a prediction about future hidden states; it can be converted into a prediction about future observations using

$$p(\mathbf{x}_{t+h}|\mathbf{x}_{1:t}) = \sum_{z_{t+h}} p(\mathbf{x}_{t+h}|z_{t+h}) p(z_{t+h}|\mathbf{x}_{1:t}) \tag{17.43}$$

  This is the posterior predictive density, and can be used for time-series forecasting (see (Fraser 2008) for details). See Figure 17.11 for a sketch of the relationship between filtering, smoothing, and prediction.
- **MAP estimation** This means computing $\arg\max_{\mathbf{z}_{1:T}} p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T})$, which is a most probable state sequence. In the context of HMMs, this is known as **Viterbi decoding** (see

Section 17.4.4). Figure 17.10 illustrates the difference between filtering, smoothing and MAP decoding for the occasionally dishonest casino HMM. We see that the smoothed (offline) estimate is indeed smoother than the filtered (online) estimate. If we threshold the estimates at 0.5 and compare to the true sequence, we find that the filtered method makes 71 errors out of 300, and the smoothed method makes 49/300; the MAP path makes 60/300 errors. It is not surprising that smoothing makes fewer errors than Viterbi, since the optimal way to minimize bit-error rate is to threshold the posterior marginals (see Section 5.7.1.1). Nevertheless, for some applications, we may prefer the Viterbi decoding, as we discuss in Section 17.4.4.

- **Posterior samples** If there is more than one plausible interpretation of the data, it can be useful to sample from the posterior, $\mathbf{z}_{1:T} \sim p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T})$. These sample paths contain much more information than the sequence of marginals computed by smoothing.
- **Probability of the evidence** We can compute the **probability of the evidence**, $p(\mathbf{x}_{1:T})$, by summing up over all hidden paths, $p(\mathbf{x}_{1:T}) = \sum_{\mathbf{z}_{1:T}} p(\mathbf{z}_{1:T}, \mathbf{x}_{1:T})$. This can be used to classify sequences (e.g., if the HMM is used as a class conditional density), for model-based clustering, for anomaly detection, etc.

17.4.2 The forwards algorithm

We now describe how to recursively compute the filtered marginals, $p(z_t|\mathbf{x}_{1:t})$ in an HMM.

The algorithm has two steps. First comes the prediction step, in which we compute the **one-step-ahead predictive density**; this acts as the new prior for time t:

$$p(z_t = j|\mathbf{x}_{1:t-1}) = \sum_i p(z_t = j|z_{t-1} = i)p(z_{t-1} = i|\mathbf{x}_{1:t-1}) \tag{17.44}$$

Next comes the update step, in which we absorb the observed data from time t using Bayes rule:

$$\alpha_t(j) \triangleq p(z_t = j|\mathbf{x}_{1:t}) = p(z_t = j|\mathbf{x}_t, \mathbf{x}_{1:t-1}) \tag{17.45}$$

$$= \frac{1}{Z_t} p(\mathbf{x}_t|z_t = j, \cancel{\mathbf{x}_{1:t-1}}) p(z_t = j|\mathbf{x}_{1:t-1}) \tag{17.46}$$

where the normalization constant is given by

$$Z_t \triangleq p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = \sum_j p(z_t = j|\mathbf{x}_{1:t-1}) p(\mathbf{x}_t|z_t = j) \tag{17.47}$$

This process is known as the **predict-update cycle**. The distribution $p(z_t|\mathbf{x}_{1:t})$ is called the (filtered) **belief state** at time t, and is a vector of K numbers, often denoted by $\boldsymbol{\alpha}_t$. In matrix-vector notation, we can write the update in the following simple form:

$$\boldsymbol{\alpha}_t \propto \boldsymbol{\psi}_t \odot (\boldsymbol{\Psi}^T \boldsymbol{\alpha}_{t-1}) \tag{17.48}$$

where $\psi_t(j) = p(\mathbf{x}_t|z_t = j)$ is the local evidence at time t, $\Psi(i, j) = p(z_t = j|z_{t-1} = i)$ is the transition matrix, and $\mathbf{u} \odot \mathbf{v}$ is the **Hadamard product**, representing elementwise vector multiplication. See Algorithm 6 for the pseudo-code, and `hmmFilter` for some Matlab code.

In addition to computing the hidden states, we can use this algorithm to compute the log probability of the evidence:

$$\log p(\mathbf{x}_{1:T}|\boldsymbol{\theta}) = \sum_{t=1}^{T} \log p(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = \sum_{t=1}^{T} \log Z_t \tag{17.49}$$

(We need to work in the log domain to avoid numerical underflow.)

Algorithm 17.1: Forwards algorithm

Input: Transition matrices $\psi(i,j) = p(z_t = j|z_{t-1} = i)$, local evidence vectors $\psi_t(j) = p(\mathbf{x}_t|z_t = j)$, initial state distribution $\pi(j) = p(z_1 = j)$;
$[\alpha_1, Z_1] = \text{normalize}(\boldsymbol{\psi}_1 \odot \boldsymbol{\pi})$;
for $t = 2 : T$ **do**
  $[\alpha_t, Z_t] = \text{normalize}(\boldsymbol{\psi}_t \odot (\boldsymbol{\Psi}^T \boldsymbol{\alpha}_{t-1}))$;
Return $\boldsymbol{\alpha}_{1:T}$ and $\log p(\mathbf{y}_{1:T}) = \sum_t \log Z_t$;
Subroutine: $[\mathbf{v}, Z] = \text{normalize}(\mathbf{u}) : Z = \sum_j u_j; \quad v_j = u_j/Z$;

17.4.3 The forwards-backwards algorithm

In Section 17.4.2, we explained how to compute the filtered marginals $p(z_t = j|\mathbf{x}_{1:t})$ using online inference. We now discuss how to compute the smoothed marginals, $p(z_t = j|\mathbf{x}_{1:T})$, using offline inference.

17.4.3.1 Basic idea

The key decomposition relies on the fact that we can break the chain into two parts, the past and the future, by conditioning on z_t:

$$p(z_t = j|\mathbf{x}_{1:T}) \propto p(z_t = j, \mathbf{x}_{t+1:T}|\mathbf{x}_{1:t}) \propto p(z_t = j|\mathbf{x}_{1:t}) p(\mathbf{x}_{t+1:T}|z_t = j, \cancel{\mathbf{x}_{1:t}}) \tag{17.50}$$

Let $\alpha_t(j) \triangleq p(z_t = j|\mathbf{x}_{1:t})$ be the filtered belief state as before. Also, define

$$\beta_t(j) \triangleq p(\mathbf{x}_{t+1:T}|z_t = j) \tag{17.51}$$

as the conditional likelihood of future evidence given that the hidden state at time t is j. (Note that this is not a probability distribution over states, since it does not need to satisfy $\sum_j \beta_t(j) = 1$.) Finally, define

$$\gamma_t(j) \triangleq p(z_t = j|\mathbf{x}_{1:T}) \tag{17.52}$$

as the desired smoothed posterior marginal. From Equation 17.50, we have

$$\gamma_t(j) \propto \alpha_t(j)\beta_t(j) \tag{17.53}$$

We have already described how to recursively compute the α's in a left-to-right fashion in Section 17.4.2. We now describe how to recursively compute the β's in a right-to-left fashion. If we have already computed β_t, we can compute β_{t-1} as follows:

$$\begin{aligned}
\beta_{t-1}(i) &= p(\mathbf{x}_{t:T}|z_{t-1}=i) && (17.54)\\
&= \sum_j p(z_t=j, \mathbf{x}_t, \mathbf{x}_{t+1:T}|z_{t-1}=i) && (17.55)\\
&= \sum_j p(\mathbf{x}_{t+1:T}|z_t=j, \cancel{z_{t-1}=i}, \cancel{\mathbf{x}_t}) p(z_t=j, \mathbf{x}_t|z_{t-1}=i) && (17.56)\\
&= \sum_j p(\mathbf{x}_{t+1:T}|z_t=j) p(\mathbf{x}_t|z_t=j, \cancel{z_{t-1}=i}) p(z_t=j|z_{t-1}=i) && (17.57)\\
&= \sum_j \beta_t(j)\psi_t(j)\psi(i,j) && (17.58)
\end{aligned}$$

We can write the resulting equation in matrix-vector form as

$$\boldsymbol{\beta}_{t-1} = \boldsymbol{\Psi}(\boldsymbol{\psi}_t \odot \boldsymbol{\beta}_t) \tag{17.59}$$

The base case is

$$\beta_T(i) = p(\mathbf{x}_{T+1:T}|z_T=i) = p(\emptyset|z_T=i) = 1 \tag{17.60}$$

which is the probability of a non-event.

Having computed the forwards and backwards messages, we can combine them to compute $\gamma_t(j) \propto \alpha_t(j)\beta_t(j)$. The overall algorithm is known as the **forwards-backwards algorithm**. The pseudo code is very similar to the forwards case; see `hmmFwdBack` for an implementation.

We can think of this algorithm as passing "messages" from left to right, and then from right to left, and then combining them at each node. We will generalize this intuition in Section 20.2, when we discuss belief propagation.

17.4.3.2 Two-slice smoothed marginals

When we estimate the parameters of the transition matrix using EM (see Section 17.5), we will need to compute the expected number of transitions from state i to state j:

$$N_{ij} = \sum_{t=1}^{T-1} \mathbb{E}\left[\mathbb{I}(z_t=i, z_{t+1}=j)|\mathbf{x}_{1:T}\right] = \sum_{t=1}^{T-1} p(z_t=i, z_{t+1}=j|\mathbf{x}_{1:T}) \tag{17.61}$$

The term $p(z_t=i, z_{t+1}=j|\mathbf{x}_{1:T})$ is called a (smoothed) **two-slice marginal**, and can be computed as follows

$$\begin{aligned}
\xi_{t,t+1}(i,j) &\triangleq p(z_t=i, z_{t+1}=j|\mathbf{x}_{1:T}) && (17.62)\\
&\propto p(z_t|\mathbf{x}_{1:t}) p(z_{t+1}|z_t, \mathbf{x}_{t+1:T}) && (17.63)\\
&\propto p(z_t|\mathbf{x}_{1:t}) p(\mathbf{x}_{t+1:T}|z_t, z_{t+1}) p(z_{t+1}|z_t) && (17.64)\\
&\propto p(z_t|\mathbf{x}_{1:t}) p(\mathbf{x}_{t+1}|z_{t+1}) p(\mathbf{x}_{t+2:T}|z_{t+1}) p(z_{t+1}|z_t) && (17.65)\\
&= \alpha_t(i)\phi_{t+1}(j)\beta_{t+1}(j)\psi(i,j) && (17.66)
\end{aligned}$$

In matrix-vector form, we have

$$\boldsymbol{\xi}_{t,t+1} \propto \boldsymbol{\Psi} \odot (\boldsymbol{\alpha}_t (\boldsymbol{\phi}_{t+1} \odot \boldsymbol{\beta}_{t+1})^T) \tag{17.67}$$

For another interpretation of these equations, see Section 20.2.4.3.

17.4.3.3 Time and space complexity

It is clear that a straightforward implementation of FB takes $O(K^2T)$ time, since we must perform a $K \times K$ matrix multiplication at each step. For some applications, such as speech recognition, K is very large, so the $O(K^2)$ term becomes prohibitive. Fortunately, if the transition matrix is sparse, we can reduce this substantially. For example, in a left-to-right transition matrix, the algorithm takes $O(TK)$ time.

In some cases, we can exploit special properties of the state space, even if the transition matrix is not sparse. In particular, suppose the states represent a discretization of an underlying continuous state-space, and the transition matrix has the form $\psi(i,j) \propto \exp(-\sigma^2|\mathbf{z}_i - \mathbf{z}_j|)$, where $\mathbf{z}_i$ is the continuous vector represented by state i. Then one can implement the forwards-backwards algorithm in $O(TK \log K)$ time. This is very useful for models with large state spaces. See Section 22.2.6.1 for details.

In some cases, the bottleneck is memory, not time. The expected sufficient statistics needed by EM are $\sum_t \xi_{t-1,t}(i,j)$; this takes constant space (independent of T); however, to compute them, we need $O(KT)$ working space, since we must store α_t for $t = 1, \ldots, T$ until we do the backwards pass. It is possible to devise a simple divide-and-conquer algorithm that reduces the space complexity from $O(KT)$ to $O(K \log T)$ at the cost of increasing the running time from $O(K^2T)$ to $O(K^2T \log T)$: see (Binder et al. 1997; Zweig and Padmanabhan 2000) for details.

17.4.4 The Viterbi algorithm

The **Viterbi** algorithm (Viterbi 1967) can be used to compute the most probable sequence of states in a chain-structured graphical model, i.e., it can compute

$$\mathbf{z}^* = \arg\max_{\mathbf{z}_{1:T}} p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T}) \tag{17.68}$$

This is equivalent to computing a shortest path through the **trellis diagram** in Figure 17.12, where the nodes are possible states at each time step, and the node and edge weights are log probabilities. That is, the weight of a path $z_1, z_2, \ldots, z_T$ is given by

$$\log \pi_1(z_1) + \log \phi_1(z_1) + \sum_{t=2}^{T} [\log \psi(z_{t-1}, z_t) + \log \phi_t(z_t)] \tag{17.69}$$

17.4.4.1 MAP vs MPE

Before discussing how the algorithm works, let us make one important remark: the *(jointly) most probable sequence of states is not necessarily the same as the sequence of (marginally) most probable states*. The former is given by Equation 17.68, and is what Viterbi computes, whereas the latter is given by the maximizer of the posterior marginals or **MPM**:

$$\hat{\mathbf{z}} = (\arg\max_{z_1} p(z_1|\mathbf{x}_{1:T}), \ldots, \arg\max_{z_T} p(z_T|\mathbf{x}_{1:T})) \tag{17.70}$$

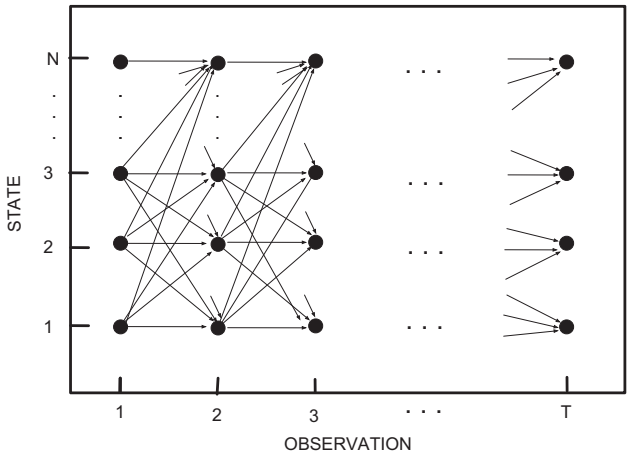


Figure 17.12 The trellis of states vs time for a Markov chain. Based on (Rabiner 1989).

As a simple example of the difference, consider a chain with two time steps, defining the following joint:

	$X_1 = 0$	$X_1 = 1$	
$X_2 = 0$	0.04	0.3	0.34
$X_2 = 1$	0.36	0.3	0.66
	0.4	0.6	

The joint MAP estimate is $(0, 1)$, whereas the sequence of marginal MPMs is $(1, 1)$.

The advantage of the joint MAP estimate is that is is always globally consistent. For example, suppose we are performing speech recognition and someones says "recognize speech". This could be mis-heard as "wreck a nice beach". Locally it may appear that "beach" is the most probable interpretation of that particular window of sound, but when we add the requirement that the data be explained by a single linguistically plausible path, this interpretation becomes less likely.

On the other hand, the MPM estimates can be more robust (Marroquin et al. 1987). To see why, note that in Viterbi, when we estimate z_t, we "max out" the other variables:

$$z_t^* = \arg\max_{z_t} \max_{\mathbf{z}_{1:t-1}, \mathbf{z}_{t+1:T}} p(\mathbf{z}_{1:t-1}, z_t, \mathbf{z}_{t+1:T} | \mathbf{x}_{1:T}) \tag{17.71}$$

whereas we when we use forwards-backwards, we sum out the other variables:

$$p(z_t | \mathbf{x}_{1:T}) = \sum_{\mathbf{z}_{1:t-1}, \mathbf{z}_{t+1:T}} p(\mathbf{z}_{1:t-1}, z_t, \mathbf{z}_{t+1:T} | \mathbf{x}_{1:T}) \tag{17.72}$$

This makes the MPM in Equation 17.70 more robust, since we estimate each node averaging over its neighbors, rather than conditioning on a specific value of its neighbors.[6]

6. In general, we may want to mix max and sum. For example, consider a joint distribution where we observe

17.4.4.2 Details of the algorithm

It is tempting to think that we can implement Viterbi by just replacing the sum-operator in forwards-backwards with a max-operator. The former is called the **sum-product**, and the latter the **max-product** algorithm. If there is a unique mode, running max-product and then computing using Equation 17.70 will give the same result as using Equation 17.68 (Weiss and Freeman 2001b), but in general, it can lead to incorrect results if there are multiple equally probably joint assignments. The reasons is that each node breaks ties independently and hence may do so in a manner that is inconsistent with its neighbors. The Viterbi algorithm is therefore not quite as simple as replacing sum with max. In particular, the forwards pass does use max-product, but the backwards pass uses a **traceback** procedure to recover the most probable path through the trellis of states. Essentially, once z_t picks its most probable state, the previous nodes condition on this event, and therefore they will break ties consistently.

In more detail, define

$$\delta_t(j) \triangleq \max_{z_1,\ldots,z_{t-1}} p(\mathbf{z}_{1:t-1}, z_t = j|\mathbf{x}_{1:t}) \tag{17.73}$$

This is the probability of ending up in state j at time t, given that we take the most probable path. The key insight is that the most probable path to state j at time t must consist of the most probable path to some other state i at time $t-1$, followed by a transition from i to j. Hence

$$\delta_t(j) = \max_i \delta_{t-1}(i)\psi(i,j)\phi_t(j) \tag{17.74}$$

We also keep track of the most likely previous state, for each possible state that we end up in:

$$a_t(j) = \operatorname*{argmax}_i \delta_{t-1}(i)\psi(i,j)\phi_t(j) \tag{17.75}$$

That is, $a_t(j)$ tells us the most likely previous state on the most probable path to $z_t = j$. We initialize by setting

$$\delta_1(j) = \pi_j \phi_1(j) \tag{17.76}$$

and we terminate by computing the most probable final state z_T^*:

$$z_T^* = \arg\max_i \delta_T(i) \tag{17.77}$$

We can then compute the most probable sequence of states using **traceback**:

$$z_t^* = a_{t+1}(z_{t+1}^*) \tag{17.78}$$

As usual, we have to worry about numerical underflow. We are free to normalize the $\boldsymbol{\delta}_t$ terms at each step; this will not affect the maximum. However, unlike the forwards-backwards case,

---

v and we want to query q; let n be the remaining nuisance variables. We define the MAP estimate as $\mathbf{x}_q^* = \arg\max_{\mathbf{x}_q} \sum_{\mathbf{x}_n} p(\mathbf{x}_q, \mathbf{x}_n|\mathbf{x}_v)$, where we max over $\mathbf{x}_q$ and sum over $\mathbf{x}_n$. By contrast, we define the **MPE** or most probable explanation as $(\mathbf{x}_q^*, \mathbf{x}_n^*) = \arg\max_{\mathbf{x}_q,\mathbf{x}_n} p(\mathbf{x}_q, \mathbf{x}_n|\mathbf{x}_v)$, where we max over both $\mathbf{x}_q$ and $\mathbf{x}_n$. This terminology is due to (Pearl 1988), although it is not widely used outside the Bayes net literatire. Obviously MAP=MPE if $n = \emptyset$. However, if $n \neq \emptyset$, then summing out the nuisance variables can give different results than maxing them out. Summing out nuisance variables is more sensible, but computationally harder, because of the need to combine max and sum operations (Lerner and Parr 2001).

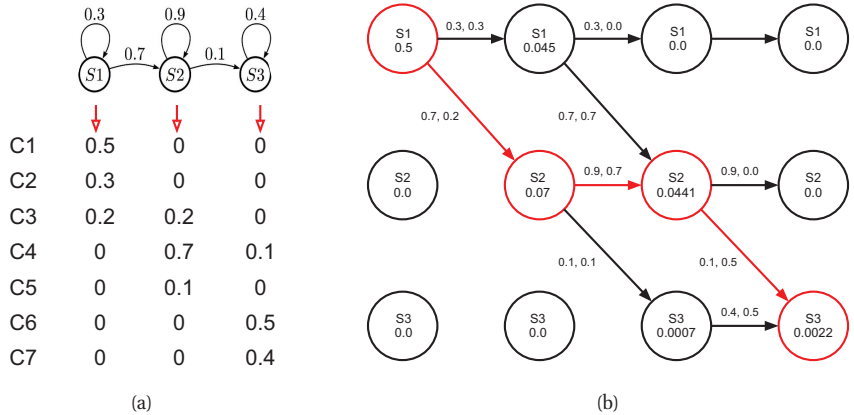


Figure 17.13 Illustration of Viterbi decoding in a simple HMM for speech recognition. (a) A 3-state HMM for a single phone. We are visualizing the state transition diagram. We assume the observations have been vector quantized into 7 possible symbols, $C_1, \ldots, C_7$. Each state z_1, z_2, z_3 has a different distribution over these symbols. Based on Figure 15.20 of (Russell and Norvig 2002). (b) Illustration of the Viterbi algorithm applied to this model, with data sequence $C1, C3, C4, C6$. The columns represent time, and the rows represent states. An arrow from state i at $t-1$ to state j at t is annotated with two numbers: the first is the probability of the $i \to j$ transition, and the second is the probability of generating observation $\mathbf{x}_t$ from state j. The bold lines/ circles represent the most probable sequence of states. Based on Figure 24.27 of (Russell and Norvig 1995).

we can also easily work in the log domain. The key difference is that $\log \max = \max \log$, whereas $\log \sum \neq \sum \log$. Hence we can use

$$\log \delta_t(j) \triangleq \max_{\mathbf{z}_{1:t-1}} \log p(\mathbf{z}_{1:t-1}, z_t = j | \mathbf{x}_{1:t}) \tag{17.79}$$

$$= \max_i \log \delta_{t-1}(i) + \log \psi(i,j) + \log \phi_t(j) \tag{17.80}$$

In the case of Gaussian observation models, this can result in a significant (constant factor) speedup, since computing $\log p(\mathbf{x}_t|z_t)$ can be much faster than computing $p(\mathbf{x}_t|z_t)$ for a high-dimensional Gaussian. This is one reason why the Viterbi algorithm is widely used in the E step of EM (Section 17.5.2) when training large speech recognition systems based on HMMs.

17.4.4.3 Example

Figure 17.13 gives a worked example of the Viterbi algorithm, based on (Russell et al. 1995). Suppose we observe the discrete sequence of observations $\mathbf{x}_{1:4} = (C_1, C_3, C_4, C_6)$, representing codebook entries in a vector-quantized version of a speech signal. The model starts in state z_1. The probability of generating C_1 in z_1 is 0.5, so we have $\delta_1(1) = 0.5$, and $\delta_1(i) = 0$ for all other states. Next we can self-transition to z_1 with probability 0.3, or transition to z_2 with proabability 0.7. If we end up in z_1, the probability of generating C_3 is 0.3; if we end up in z_2,

the probability of generating C_3 is 0.2. Hence we have

$$\delta_2(1) = \delta_1(1)\psi(1,1)\phi_2(1) = 0.5 \cdot 0.3 \cdot 0.3 = 0.045 \quad (17.81)$$

$$\delta_2(2) = \delta_1(1)\psi(1,2)\phi_2(2) = 0.5 \cdot 0.7 \cdot 0.2 = 0.07 \quad (17.82)$$

Thus state 2 is more probable at $t = 2$; see the second column of Figure 17.13(b). In time step 3, we see that there are two paths into z_2, from z_1 and from z_2. The bold arrow indicates that the latter is more probable. Hence this is the only one we have to remember. The algorithm continues in this way until we have reached the end of the sequence. One we have reached the end, we can follow the black arrows back to recover the MAP path (which is 1,2,2,3).

17.4.4.4 Time and space complexity

The time complexity of Viterbi is clearly $O(K^2T)$ in general, and the space complexity is $O(KT)$, both the same as forwards-backwards. If the transition matrix has the form $\psi(i,j) \propto \exp(-\sigma^2||\mathbf{z}_i - \mathbf{z}_j||^2)$, where $\mathbf{z}_i$ is the continuous vector represented by state i, we can implement Viterbi in $O(TK)$ time, instead of $O(TK \log K)$ needed by forwards-backwards. See Section 22.2.6.1 for details.

17.4.4.5 N-best list

The Viterbi algorithm returns one of the most probable paths. It can be extended to return the top N paths (Schwarz and Chow 1990; Nilsson and Goldberger 2001). This is called the **N-best list**. Once can then use a discriminative method to rerank the paths based on global features derived from the fully observed state sequence (as well as the visible features). This technique is widely used in speech recognition. For example, consider the sentence "recognize speech". It is possible that the most probable interpretation by the system of this acoustic signal is "wreck a nice speech", or maybe "wreck a nice beach". Maybe the correct interpretation is much lower down on the list. However, by using a re-ranking system, we may be able to improve the score of the correct interpretation based on a more global context.

One problem with the N-best list is that often the top N paths are very similar to each other, rather than representing qualitatively different interpretations of the data. Instead we might want to generate a more diverse set of paths to more accurately represent posterior uncertainty. One way to do this is to sample paths from the posterior, as we discuss below. For some other ways to generate diverse MAP estimates, see e.g., (Yadollahpour et al. 2011; Kulesza and Taskar 2011).

17.4.5 Forwards filtering, backwards sampling

It is often useful to sample paths from the posterior:

$$\mathbf{z}_{1:T}^s \sim p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T}) \quad (17.83)$$

We can do this is as follow: run forwards backwards, to compute the two-slice smoothed posteriors, $p(z_{t-1,t}|\mathbf{x}_{1:T})$; next compute the conditionals $p(z_t|z_{t-1}, \mathbf{x}_{1:T})$ by normalizing; sample from the initial pair of states, $z_{1,2}^* \sim p(z_{1,2}|\mathbf{x}_{1:T})$; finally, recursively sample $z_t^* \sim p(z_t|z_{t-1}^*, \mathbf{x}_{1:T})$.

Note that the above solution requires a forwards-backwards pass, and then an additional forwards sampling pass. An alternative is to do the forwards pass, and then perform sampling

in the backwards pass. The key insight into how to do this is that we can write the joint from right to left using

$$p(\mathbf{z}_{1:T}|\mathbf{x}_{1:T}) = p(z_T|\mathbf{x}_{1:T}) \prod_{t=T-1}^{1} p(z_t|z_{t+1}, \mathbf{x}_{1:T}) \tag{17.84}$$

We can then sample z_t given future sampled states using

$$z_t^s \sim p(z_t|z_{t+1:T}, \mathbf{x}_{1:T}) = p(z_t|z_{t+1}, \cancel{\mathbf{z}_{t+2:T}}, \mathbf{x}_{1:t}, \cancel{\mathbf{x}_{t+1:T}}) = p(z_t|z_{t+1}^s, \mathbf{x}_{1:t}) \tag{17.85}$$

The sampling distribution is given by

$$\begin{aligned} p(z_t = i|z_{t+1} = j, \mathbf{x}_{1:t}) &= p(z_t|z_{t+1}, \mathbf{x}_{1:t}, \cancel{\mathbf{x}_{t+1}}) && (17.86)\\ &= \frac{p(z_{t+1}, z_t|\mathbf{x}_{1:t+1})}{p(z_{t+1}|\mathbf{x}_{1:t+1})} && (17.87)\\ &\propto \frac{p(\mathbf{x}_{t+1}|z_{t+1}, \cancel{z_t}, \cancel{\mathbf{x}_{1:t}})p(z_{t+1}, z_t|\mathbf{x}_{1:t})}{p(z_{t+1}|\mathbf{x}_{1:t+1})} && (17.88)\\ &= \frac{p(\mathbf{x}_{t+1}|z_{t+1})p(z_{t+1}|z_t, \cancel{\mathbf{x}_{1:t}})p(z_t|\mathbf{x}_{1:t})}{p(z_{t+1}|\mathbf{x}_{1:t+1})} && (17.89)\\ &= \frac{\phi_{t+1}(j)\psi(i,j)\alpha_t(i)}{\alpha_{t+1}(j)} && (17.90) \end{aligned}$$

The base case is

$$z_T^s \sim p(z_T = i|\mathbf{x}_{1:T}) = \alpha_T(i) \tag{17.91}$$

This algorithm forms the basis of blocked-Gibbs sampling methods for parameter inference, as we will see below.

17.5 Learning for HMMs

We now discuss how to estimate the parameters $\boldsymbol{\theta} = (\boldsymbol{\pi}, \mathbf{A}, \mathbf{B})$, where $\pi(i) = p(z_1 = i)$ is the initial state distribution, $A(i,j) = p(z_t = j|z_{t-1} = i)$ is the transition matrix, and $\mathbf{B}$ are the parameters of the class-conditional densities $p(\mathbf{x}_t|z_t = j)$. We first consider the case where $\mathbf{z}_{1:T}$ is observed in the training set, and then the harder case where $\mathbf{z}_{1:T}$ is hidden.

17.5.1 Training with fully observed data

If we observe the hidden state sequences, we can compute the MLEs for $\mathbf{A}$ and $\boldsymbol{\pi}$ exactly as in Section 17.2.2.1. If we use a conjugate prior, we can also easily compute the posterior.

The details on how to estimate $\mathbf{B}$ depend on the form of the observation model. The situation is identical to fitting a generative classifier. For example, if each state has a multinoulli distribution associated with it, with parameters $B_{jl} = p(X_t = l|z_t = j)$, where $l \in \{1, \ldots, L\}$ represents the observed symbol, the MLE is given by

$$\hat{B}_{jl} = \frac{N_{jl}^X}{N_j}, \quad N_{jl}^X \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = j, x_{i,t} = l) \tag{17.92}$$

This result is quite intuitive: we simply add up the number of times we are in state j and we see a symbol l, and divide by the number of times we are in state j.

Similarly, if each state has a Gaussian distribution associated with it, we have (from Section 4.2.4) the following MLEs:

$$\hat{\boldsymbol{\mu}}_k = \frac{\overline{\mathbf{x}}_k}{N_k}, \;\; \hat{\boldsymbol{\Sigma}}_k = \frac{(\overline{\mathbf{xx}})_k^T - N_k \hat{\boldsymbol{\mu}}_k \hat{\boldsymbol{\mu}}_k^T}{N_k} \tag{17.93}$$

where the sufficient statistics are given by

$$\overline{\mathbf{x}}_k \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = k) \mathbf{x}_{i,t} \tag{17.94}$$

$$(\overline{\mathbf{xx}})_k^T \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = k) \mathbf{x}_{i,t} \mathbf{x}_{i,t}^T \tag{17.95}$$

Analogous results can be derived for other kinds of distributions. One can also easily extend all of these results to compute MAP estimates, or even full posteriors over the parameters.

17.5.2 EM for HMMs (the Baum-Welch algorithm)

If the z_t variables are not observed, we are in a situation analogous to fitting a mixture model. The most common approach is to use the EM algorithm to find the MLE or MAP parameters, although of course one could use other gradient-based methods (see e.g., (Baldi and Chauvin 1994)). In this Section, we derive the EM algorithm. When applied to HMMs, this is also known as the **Baum-Welch** algorithm (Baum et al. 1970).

17.5.2.1 E step

It is straightforward to show that the expected complete data log likelihood is given by

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{old}) = \sum_{k=1}^{K} \mathbb{E}\left[N_k^1\right] \log \pi_k + \sum_{j=1}^{K} \sum_{k=1}^{K} \mathbb{E}\left[N_{jk}\right] \log A_{jk} \tag{17.96}$$

$$+ \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{k=1}^{K} p(z_t = k | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \log p(\mathbf{x}_{i,t} | \boldsymbol{\phi}_k) \tag{17.97}$$

where the expected counts are given by

$$\mathbb{E}\left[N_k^1\right] = \sum_{i=1}^{N} p(z_{i1} = k | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \tag{17.98}$$

$$\mathbb{E}\left[N_{jk}\right] = \sum_{i=1}^{N} \sum_{t=2}^{T_i} p(z_{i,t-1} = j, z_{i,t} = k | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \tag{17.99}$$

$$\mathbb{E}\left[N_j\right] = \sum_{i=1}^{N} \sum_{t=1}^{T_i} p(z_{i,t} = j | \mathbf{x}_i, \boldsymbol{\theta}^{old}) \tag{17.100}$$

These expected sufficient statistics can be computed by running the forwards-backwards algorithm on each sequence. In particular, this algorithm computes the following smoothed node and edge marginals:

$$\gamma_{i,t}(j) \triangleq p(z_t = j|\mathbf{x}_{i,1:T_i}, \boldsymbol{\theta}) \tag{17.101}$$

$$\xi_{i,t}(j,k) \triangleq p(z_{t-1} = j, z_t = k|\mathbf{x}_{i,1:T_i}, \boldsymbol{\theta}) \tag{17.102}$$

17.5.2.2 M step

Based on Section 11.3, we have that the M step for $\mathbf{A}$ and $\boldsymbol{\pi}$ is to just normalize the expected counts:

$$\hat{A}_{jk} = \frac{\mathbb{E}\left[N_{jk}\right]}{\sum_{k'} \mathbb{E}\left[N_{jk'}\right]}, \; \hat{\pi}_k = \frac{\mathbb{E}\left[N_k^1\right]}{N} \tag{17.103}$$

This result is quite intuitive: we simply add up the expected number of transitions from j to k, and divide by the expected number of times we transition from j to anything else.

For a multinoulli observation model, the expected sufficient statistics are

$$\mathbb{E}\left[M_{jl}\right] = \sum_{i=1}^{N}\sum_{t=1}^{T_i} \gamma_{i,t}(j)\mathbb{I}(x_{i,t} = l) = \sum_{i=1}^{N}\sum_{t:x_{i,t}=l} \gamma_{i,t}(j) \tag{17.104}$$

The M step has the form

$$\hat{B}_{jl} = \frac{\mathbb{E}\left[M_{jl}\right]}{\mathbb{E}\left[N_j\right]} \tag{17.105}$$

This result is quite intuitive: we simply add up the expected number of times we are in state j and we see a symbol l, and divide by the expected number of times we are in state j.

For a Gaussian observation model, the expected sufficient statistics are given by

$$\mathbb{E}\left[\overline{\mathbf{x}}_k\right] = \sum_{i=1}^{N}\sum_{t=1}^{T_i} \gamma_{i,t}(k)\mathbf{x}_{i,t} \tag{17.106}$$

$$\mathbb{E}\left[(\overline{\mathbf{x}\mathbf{x}})_k^T\right] = \sum_{i=1}^{N}\sum_{t=1}^{T_i} \gamma_{i,t}(k)\mathbf{x}_{i,t}\mathbf{x}_{i,t}^T \tag{17.107}$$

The M step becomes

$$\hat{\boldsymbol{\mu}}_k = \frac{\mathbb{E}\left[\overline{\mathbf{x}}_k\right]}{\mathbb{E}\left[N_k\right]}, \; \hat{\boldsymbol{\Sigma}}_k = \frac{\mathbb{E}\left[(\overline{\mathbf{x}\mathbf{x}})_k^T\right] - \mathbb{E}\left[N_k\right]\hat{\boldsymbol{\mu}}_k\hat{\boldsymbol{\mu}}_k^T}{\mathbb{E}\left[N_k\right]} \tag{17.108}$$

This can (and should) be regularized in the same way we regularize GMMs.

17.5.2.3 Initialization

As usual with EM, we must take care to ensure that we initialize the parameters carefully, to minimize the chance of getting stuck in poor local optima. There are several ways to do this, such as

- Use some fully labeled data to initialize the parameters.
- Initially ignore the Markov dependencies, and estimate the observation parameters using the standard mixture model estimation methods, such as K-means or EM.
- Randomly initialize the parameters, use multiple restarts, and pick the best solution.

Techniques such as deterministic annealing (Ueda and Nakano 1998; Rao and Rose 2001) can help mitigate the effect of local minima. Also, just as K-means is often used to initialize EM for GMMs, so it is common to initialize EM for HMMs using **Viterbi training**, which means approximating the posterior over paths with the single most probable path. (This is not necessarily a good idea, since initially the parameters are often poorly estimated, so the Viterbi path will be fairly arbitrary. A safer option is to start training using forwards-backwards, and to switch to Viterbi near convergence.)

17.5.3 Bayesian methods for "fitting" HMMs *

EM returns a MAP estimate of the parameters. In this section, we briefly discuss some methods for Bayesian parameter estimation in HMMs. (These methods rely on material that we will cover later in the book.)

One approach is to use variational Bayes EM (VBEM), which we discuss in general terms in Section 21.6. The details for the HMM case can be found in (MacKay 1997; Beal 2003), but the basic idea is this: The E step uses forwards-backwards, but where (roughly speaking) we plug in the posterior mean parameters instead of the MAP estimates. The M step updates the parameters of the conjugate posteriors, instead of updating the parameters themselves.

An alternative to VBEM is to use MCMC. A particularly appealing algorithm is block Gibbs sampling, which we discuss in general terms in Section 24.2.8. The details for the HMM case can be found in (Fruhwirth-Schnatter 2007), but the basic idea is this: we sample $\mathbf{z}_{1:T}$ given the data and parameters using forwards-filtering, backwards-sampling, and we then sample the parameters from their posteriors, conditional on the sampled latent paths. This is simple to implement, but one does need to take care of unidentifiability (label switching), just as with mixture models (see Section 11.3.1).

17.5.4 Discriminative training

Sometimes HMMs are used as the class conditional density inside a generative classifier. In this case, $p(\mathbf{x}|y=c,\boldsymbol{\theta})$ can be computed using the forwards algorithm. We can easily maximize the joint likelihood $\prod_{i=1}^{N} p(\mathbf{x}_i, y_i|\boldsymbol{\theta})$ by using EM (or some other method) to fit the HMM for each class-conditional density separately.

However, we might like to find the parameters that maximize the conditional likelihood

$$\prod_{i=1}^{N} p(y_i|\mathbf{x}_i,\boldsymbol{\theta}) = \prod_i \frac{p(y_i|\boldsymbol{\theta})p(\mathbf{x}_i|y_i,\boldsymbol{\theta})}{\sum_c p(y_i=c|\boldsymbol{\theta})p(\mathbf{x}_i|c,\boldsymbol{\theta})} \tag{17.109}$$

This is more expensive than maximizing the joint likelihood, since the denominator couples all C class-conditional HMMs together. Furthermore, EM can no longer be used, and one must resort

to generic gradient based methods. Nevertheless, discriminative training can result in improved accuracies. The standard practice in speech recognition is to initially train the generative models separately using EM, and then to fine tune them discriminatively (Jelinek 1997).

17.5.5 Model selection

In HMMs, the two main model selection issues are: how many states, and what topology to use for the state transition diagram. We discuss both of these issues below.

17.5.5.1 Choosing the number of hidden states

Choosing the number of hidden states K in an HMM is analogous to the problem of choosing the number of mixture components. Here are some possible solutions:

- Use grid-search over a range of K's, using as an objective function cross-validated likelihood, the BIC score, or a variational lower bound to the log-marginal likelihood.
- Use reversible jump MCMC. See (Fruhwirth-Schnatter 2007) for details. Note that this is very slow and is not widely used.
- Use variational Bayes to "extinguish" unwanted components, by analogy to the GMM case discussed in Section 21.6.1.6. See (MacKay 1997; Beal 2003) for details.
- Use an "infinite HMM", which is based on the hierarchical Dirichlet process. See e.g., (Beal et al. 2002; Teh et al. 2006) for details.

17.5.5.2 Structure learning

The term **structure learning** in the context of HMMs refers to learning a sparse transition matrix. That is, we want to learn the structure of the state transition diagram, not the structure of the graphical model (which is fixed). A large number of heuristic methods have been proposed. Most alternate between parameter estimation and some kind of heuristic **split merge** method (see e.g., (Stolcke and Omohundro 1992)).

Alternatively, one can pose the problem as MAP estimation using a **minimum entropy prior**, of the form

$$p(\mathbf{A}_{i,:}) \propto \exp(-\mathbb{H}(\mathbf{A}_{i,:})) \tag{17.110}$$

This prior prefers states whose outgoing distribution is nearly deterministic, and hence has low entropy (Brand 1999). The corresponding M step cannot be solved in closed form, but numerical methods can be used. The trouble with this is that we might prune out all incoming transitions to a state, creating isolated "islands" in state-space. The infinite HMM presents an interesting alternative to these methods. See e.g., (Beal et al. 2002; Teh et al. 2006) for details.

17.6 Generalizations of HMMs

Many variants of the basic HMM model have been proposed. We briefly discuss some of them below.

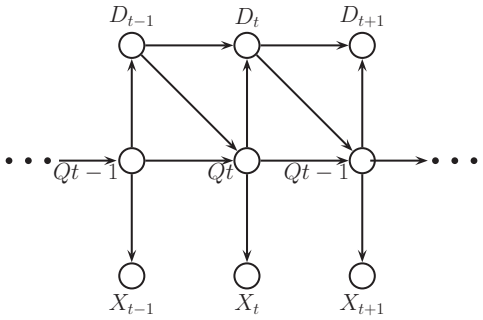


Figure 17.14 Encoding a hidden semi-Markov model as a DGM. D_t are deterministic duration counters.

17.6.1 Variable duration (semi-Markov) HMMs

In a standard HMM, the probability we remain in state i for exactly d steps is

$$p(t_i = d) = (1 - A_{ii})A_{ii}^d \propto \exp(d \log A_{ii}) \tag{17.111}$$

where A_{ii} is the self-loop probability. This is called the **geometric distribution**. However, this kind of exponentially decaying function of d is sometimes unrealistic.

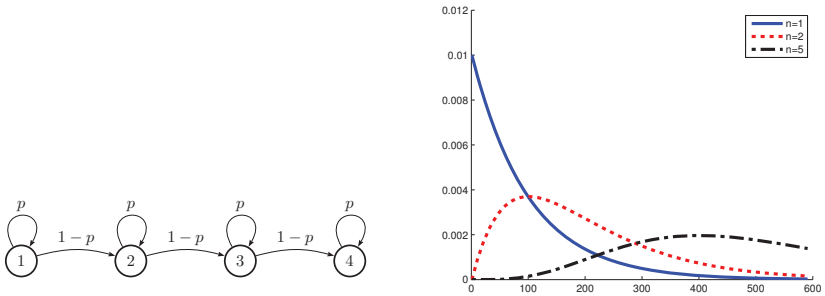
To allow for more general durations, one can use a **semi-Markov model**. It is called semi-Markov because to predict the next state, it is not sufficient to condition on the past state: we also need to know how long we've been in that state. When the state space is not observed directly, the result is called a **hidden semi-Markov model** (**HSMM**), a **variable duration HMM**, or an **explicit duration HMM**.

HSMMs are widely used in many gene finding programs, since the length distribution of exons and introns is not geometric (see e.g., (Schweikerta et al. 2009)), and in some chip-Seq data analysis programs (see e.g., (Kuan et al. 2009)).

HSMMs are useful not only because they can model the waiting time of each state more accurately, but also because they can model the distribution of a whole batch of observations at once, instead of assuming all observations are conditionally iid. That is, they can use likelihood models of the form $p(\mathbf{x}_{t:t+l}|z_t = k, d_t = l)$, which generate l correlated observations if the duration in state k is for l time steps. This is useful for modeling data that is piecewise linear, or shows other local trends (Ostendorf et al. 1996).

17.6.1.1 HSMM as augmented HMMs

One way to represent a HSMM is to use the graphical model shown in Figure 17.14. (In this figure, we have assumed the observations are iid within each state, but this is not required, as mentioned above.) The $D_t \in \{0, 1, \ldots, D\}$ node is a state duration counter, where D is the maximum duration of any state. When we first enter state j, we sample D_t from the duration distribution for that state, $D_t \sim p_j(\cdot)$. Thereafer, D_t deterministically counts down



(a) (b)

Figure 17.15 (a) A Markov chain with $n = 4$ repeated states and self loops. (b) The resulting distribution over sequence lengths, for $p = 0.99$ and various n. Figure generated by `hmmSelfLoopDist`.

until $D_t = 0$. While $D_t > 0$, the state z_t is not allowed to change. When $D_t = 0$, we make a stochastic transition to a new state.

More precisely, we define the CPDs as follows:

$$p(D_t = d'|D_{t-1} = d, z_t = j) = \begin{cases} p_j(d') & \text{if } d = 0 \\ 1 & \text{if } d' = d - 1 \text{ and } d \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{17.112}$$

$$p(z_t = k|z_{t-1} = j, D_{t-1} = d) = \begin{cases} 1 & \text{if } d > 0 \text{ and } j = k \\ A_{jk} & \text{if } d = 0 \\ 0 & \text{otherwise} \end{cases} \tag{17.113}$$

Note that $p_j(d)$ could be represented as a table (a non-parametric approach) or as some kind of parametric distribution, such as a Gamma distribution. If $p_j(d)$ is a geometric distribution, this emulates a standard HMM.

One can perform inference in this model by defining a mega-variable $Y_t = (D_t, z_t)$. However, this is rather inefficient, since D_t is deterministic. It is possible to marginalize D_t out, and derive special purpose inference procedures. See (Guedon 2003; Yu and Kobayashi 2006) for details. Unfortunately, all these methods take $O(TK^2D)$ time, where T is the sequence length, K is the number of states, and D is the maximum duration of any state.

17.6.1.2 Approximations to semi-Markov models

A more efficient, but less flexible, way to model non-geometric waiting times is to replace each state with n new states, each with the same emission probabilities as the original state. For example, consider the model in Figure 17.15(a). Obviously the smallest sequence this can generate is of length $n = 4$. Any path of length d through the model has probability $p^{d-n}(1-p)^n$; multiplying by the number of possible paths we find that the total probability of a path of length d is

$$p(d) = \binom{d-1}{n-1} p^{d-n}(1-p)^n \tag{17.114}$$

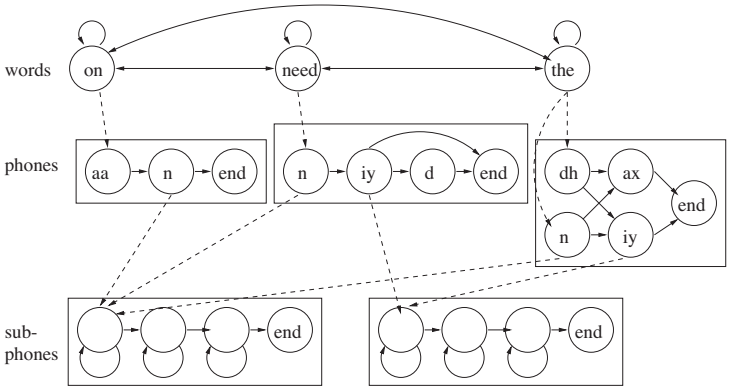


Figure 17.16 An example of an HHMM for an ASR system which can recognize 3 words. The top level represents bigram word probabilities. The middle level represents the phonetic spelling of each word. The bottom level represents the subphones of each phone. (It is traditional to represent a phone as a 3 state HMM, representing the beginning, middle and end.) Based on Figure 7.5 of (Jurafsky and Martin 2000).

This is equivalent to the negative binomial distribution. By adjusting n and the self-loop probabilities p of each state, we can model a wide range of waiting times: see Figure 17.15(b).

Let E be the number of expansions of each state needed to approximate $p_j(d)$. Forwards-backwards on this model takes $O(T(KE)F_{in})$ time, where F_{in} is the average number of predecessor states, compared to $O(TK(F_{in}+D))$ for the HSMM. For typical speech recognition applications, $F_{in} \sim 3$, $D \sim 50$, $K \sim 10^6$, $T \sim 10^5$. (Similar figures apply to problems such as gene finding, which also often uses HSMMs.) Since $F_{in} + D \gg EF_{in}$, the expanded state method is much faster than an HSMM. See (Johnson 2005) for details.

17.6.2 Hierarchical HMMs

A **hierarchical HMM** (HHMM) (Fine et al. 1998) is an extension of the HMM that is designed to model domains with hierarchical structure. Figure 17.16 gives an example of an HHMM used in automatic speech recognition. The phone and subphone models can be "called" from different higher level contexts. We can always "flatten" an HHMM to a regular HMM, but a factored representation is often easier to interpret, and allows for more efficient inference and model fitting.

HHMMs have been used in many application domains, e.g., speech recognition (Bilmes 2001), gene finding (Hu et al. 2000), plan recognition (Bui et al. 2002), monitoring transportation patterns (Liao et al. 2007), indoor robot localization (Theocharous et al. 2004), etc. HHMMs are less expressive than stochastic context free grammars (SCFGs), since they only allow hierarchies of bounded depth, but they support more efficient inference. In particular, inference in SCFGs (using the inside outside algorithm, (Jurafsky and Martin 2008)) takes $O(T^3)$ whereas inference in an HHMM takes $O(T)$ time (Murphy and Paskin 2001).

We can represent an HHMM as a directed graphical model as shown in Figure 17.17. Q_t^ℓ represents the state at time t and level ℓ. A state transition at level ℓ is only "allowed" if the

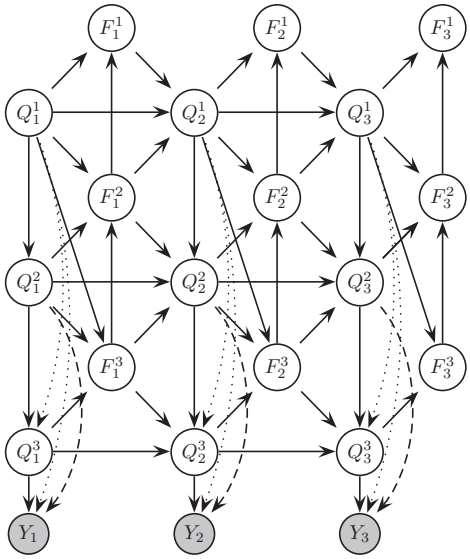


Figure 17.17 An HHMM represented as a DGM. Q_t^ℓ is the state at time t, level ℓ; $F_t^\ell = 1$ if the HMM at level ℓ has finished (entered its exit state), otherwise $F_t^\ell = 0$. Shaded nodes are observed; the remaining nodes are hidden. We may optionally clamp $F_T^\ell = 1$, where T is the length of the observation sequence, to ensure all models have finished by the end of the sequence. Source: Figure 2 of (Murphy and Paskin 2001).

chain at the level below has "finished", as determined by the $F_t^{\ell-1}$ node. (The chain below finishes when it chooses to enter its end state.) This mechanism ensures that higher level chains evolve more slowly than lower level chains, i.e., lower levels are nested within higher levels.

A variable duration HMM can be thought of as a special case of an HHMM, where the top level is a deterministic counter, and the bottom level is a regular HMM, which can only change states once the counter has "timed out". See (Murphy and Paskin 2001) for further details.

17.6.3 Input-output HMMs

It is straightforward to extend an HMM to handle inputs, as shown in Figure 17.18(a). This defines a conditional density model for sequences of the form

$$p(\mathbf{y}_{1:T}, \mathbf{z}_{1:T}|\mathbf{u}_{1:T}, \boldsymbol{\theta}) \tag{17.115}$$

where $\mathbf{u}_t$ is the input at time t; this is sometimes called a control signal. If the inputs and outputs are continuous, a typical parameterization would be

$$p(z_t|\mathbf{x}_t, z_{t-1} = i, \boldsymbol{\theta}) = \mathrm{Cat}(z_t|\mathcal{S}(\mathbf{W}_i\mathbf{u}_t)) \tag{17.116}$$

$$p(\mathbf{y}_t|\mathbf{x}_t, z_t = j, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}_t|\mathbf{V}_j\mathbf{u}_t, \boldsymbol{\Sigma}_j) \tag{17.117}$$

Thus the transition matrix is a logistic regression model whose parameters depend on the previous state. The observation model is a Gaussian whose parameters depend on the current

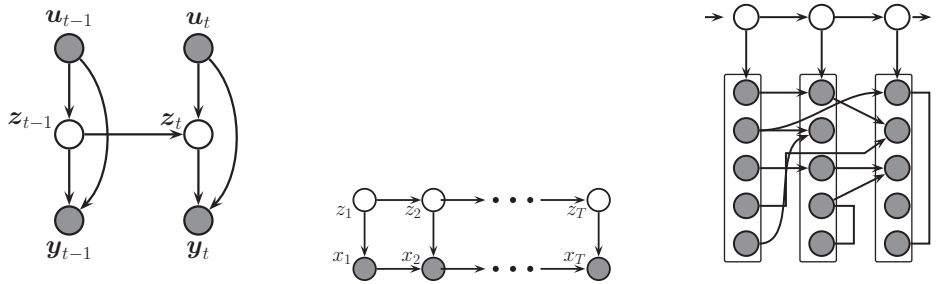


Figure 17.18 (a) Input-output HMM. (b) First-order auto-regressive HMM. (c) A second-order buried Markov model. Depending on the value of the hidden variables, the effective graph structure between the components of the observed variables (i.e., the non-zero elements of the regression matrix and the precision matrix) can change, although this is not shown.

state. The whole model can be thought of as a hidden version of a maximum entropy Markov model (Section 19.6.1).

Conditional on the inputs $\mathbf{u}_{1:T}$ and the parameters $\boldsymbol{\theta}$, one can apply the standard forwards-backwards algorithm to estimate the hidden states. It is also straightforward to derive an EM algorithm to estimate the parameters (see (Bengio and Frasconi 1996) for details).

17.6.4 Auto-regressive and buried HMMs

The standard HMM assumes the observations are conditionally independent given the hidden state. In practice this is often not the case. However, it is straightforward to have direct arcs from $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$ as well as from z_t to $\mathbf{x}_t$, as in Figure 17.18(b). This is known as an **auto-regressive HMM**, or a **regime switching Markov model**. For continuous data, the observation model becomes

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, z_t = j, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\mathbf{W}_j\mathbf{x}_{t-1} + \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{17.118}$$

This is a linear regression model, where the parameters are chosen according to the current hidden state. We can also consider higher-order extensions, where we condition on the last L observations:

$$p(\mathbf{x}_t|\mathbf{x}_{t-L:t-1}, z_t = j, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\sum_{\ell=1}^{L}\mathbf{W}_{j,\ell}\mathbf{x}_{t-\ell} + \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{17.119}$$

Such models are widely used in econometrics (Hamilton 1990). Similar models can be defined for discrete observations.

The AR-HMM essentially combines two Markov chains, one on the hidden variables, to capture long range dependencies, and one on the observed variables, to capture short range dependencies (Berchtold 1999). Since the X nodes are observed, the connections between them only

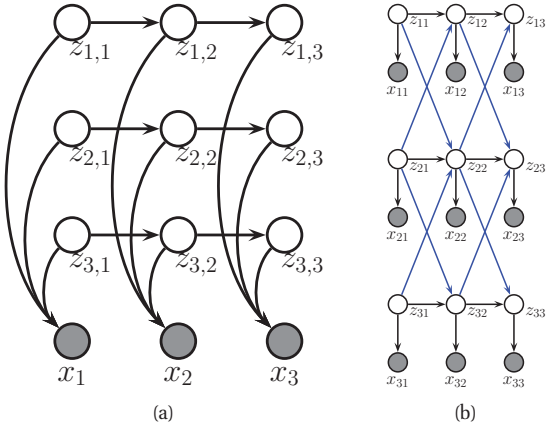


Figure 17.19 (a) A factorial HMM with 3 chains. (b) A coupled HMM with 3 chains.

change the computation of the local evidence; inference can still be performed using the standard forwards-backwards algorithm. Parameter estimation using EM is also straightforward: the E step is unchanged, as is the M step for the transition matrix. If we assume scalar observations for notational simplicty, the M step involves minimizing

$$\sum_t \mathbb{E}\left[\frac{1}{\sigma^2(s_t)}(y_t - \mathbf{y}_{t-L:t-1}^T \mathbf{w}(s_t))^2 + \log \sigma^2(s_t)\right] \tag{17.120}$$

Focussing on the $\mathbf{w}$ terms, we see that this requires solving K weighted least squares problems:

$$J(\mathbf{w}_{1:K}) = \sum_j \sum_t \frac{\gamma_t(j)}{\sigma^2(j)}(y_t - \mathbf{y}_{t-L:t-1}^T \mathbf{w}_j)^2 \tag{17.121}$$

where $\gamma_t(j) = p(z_t = k|\mathbf{x}_{1:T})$ is the smoothed posterior marginal. This is a weighted linear regression problem, where the design matrix has a Toeplitz form. This subproblem can be solved efficiently using the Levinson-Durbin method (Durbin and Koopman 2001).

Buried Markov models generalize AR-HMMs by allowing the dependency structure between the observable nodes to change based on the hidden state, as in Figure 17.18(c). Such a model is called a dynamic Bayesian **multi net**, since it is a mixture of different networks. In the linear-Gaussian setting, we can change the structure of the of $\mathbf{x}_{t-1} \rightarrow \mathbf{x}_t$ arcs by using sparse regression matrices, $\mathbf{W}_j$, and we can change the structure of the connections within the components of $\mathbf{x}_t$ by using sparse Gaussian graphical models, either directed or undirected. See (Bilmes 2000) for details.

17.6.5 Factorial HMM

An HMM represents the hidden state using a single discrete random variable $z_t \in \{1, \ldots, K\}$. To represent 10 bits of information would require $K = 2^{10} = 1024$ states. By contrast, consider a **distributed representation** of the hidden state, where each $z_{c,t} \in \{0, 1\}$ represents the c'th

bit of the t'th hidden state. Now we can represent 10 bits using just 10 binary variables, as illustrated in Figure 17.19(a). This model is called a **factorial HMM** (Ghahramani and Jordan 1997). The hope is that this kind of model could capture different aspects of a signal, e.g., one chain would represent speaking style, another the words that are being spoken.

Unfortunately, conditioned on $\mathbf{x}_t$, all the hidden variables are correlated (due to explaining away the common observed child $\mathbf{x}_t$). This make exact state estimation intractable. However, we can derive efficient approximate inference algorithms, as we discuss in Section 21.4.1.

17.6.6 Coupled HMM and the influence model

If we have multiple related data streams, we can use a **coupled HMM** (Brand 1996), as illustrated in Figure 17.19(b). This is a series of HMMs where the state transitions depend on the states of neighboring chains. That is, we represent the joint conditional distribution as

$$p(\mathbf{z}_t|\mathbf{z}_{t-1}) = \prod_c p(z_{ct}|\mathbf{z}_{t-1}) \tag{17.122}$$

$$p(z_{ct}|\mathbf{z}_{t-1}) = p(z_{ct}|z_{c,t-1},\ z_{c-1,t-1},\ z_{c+1,t-1}) \tag{17.123}$$

This has been used for various tasks, such as **audio-visual speech recognition** (Nefian et al. 2002) and modeling freeway traffic flows (Kwon and Murphy 2000).

The trouble with the above model is that it requires $O(CK^4)$ parameters to specify, if there are C chains with K states per chain, because each state depends on its own past plus the past of its two neighbors. There is a closely related model, known as the **influence model** (Asavathiratham 2000), which uses fewer parameters. It models the joint conditional distribution as

$$p(z_{ct}|\mathbf{z}_{t-1}) = \sum_{c'=1}^{C} \alpha_{c,c'} p(z_{ct}|z_{c',t-1}) \tag{17.124}$$

where $\sum_{c'} \alpha_{c,c'} = 1$ for each c. That is, we use a convex combination of pairwise transition matrices. The $\alpha_{c,c'}$ parameter specifies how much influence chain c has on chain c'. This model only takes $O(C^2 + CK^2)$ parameters to specify. Furthermore, it allows each chain to be influenced by all the other chains, not just its nearest neighbors. (Hence the corresponding graphical model is similar to Figure 17.19(b), except that each node has incoming edges from all the previous nodes.) This has been used for various tasks, such as modeling conversational interactions between people (Basu et al. 2001).

Unfortunately, inference in both of these models takes $O(T(K^C)^2)$ time, since all the chains become fully correlated even if the interaction graph is sparse. Various approximate inference methods can be applied, as we discuss later.

17.6.7 Dynamic Bayesian networks (DBNs)

A **dynamic Bayesian network** is just a way to represent a stochastic process using a directed graphical model.[7] Note that the network is not dynamic (the structure and parameters are fixed),

7. The acronym **DBN** can stand for either "dynamic Bayesian network" or "deep belief network" (Section 28.1) depending on the context. Geoff Hinton (who invented the term "deep belief network") has suggested the acronyms **DyBN** and **DeeBN** to avoid this ambiguity.

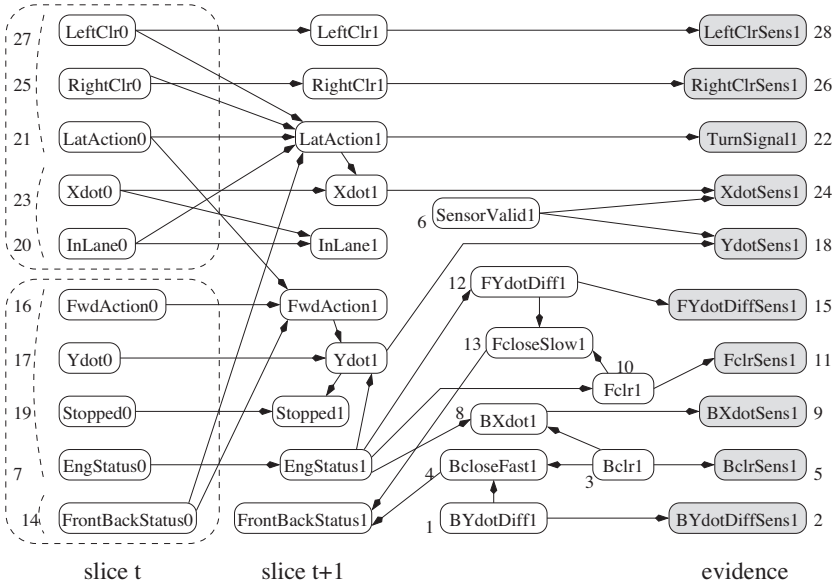


Figure 17.20 The BATnet DBN. The transient nodes are only shown for the second slice, to minimize clutter. The dotted lines can be ignored. Used with kind permission of Daphne Koller.

rather it is a network representation of a dynamical system. All of the HMM variants we have seen above could be considered to be DBNs. However, we prefer to reserve the term "DBN" for graph structures that are more "irregular" and problem-specific. An example is shown in Figure 17.20, which is a DBN designed to monitor the state of a simulated autonomous car known as the "Bayesian Automated Taxi", or "BATmobile" (Forbes et al. 1995).

Defining DBNs is straightforward: you just need to specify the structure of the first time-slice, the structure between two time-slices, and the form of the CPDs. Learning is also easy. The main problem is that exact inference can be computationally expensive, because all the hidden variables become correlated over time (this is known as **entanglement** — see e.g., (Koller and Friedman 2009, Sec. 15.2.4) for details). Thus a sparse graph does not necessarily result in tractable exact inference. However, later we will see algorithms that can exploit the graph structure for efficient approximate inference.

Exercises

Exercise 17.1 Derivation of Q function for HMM

Derive Equation 17.97.

Exercise 17.2 Two filter approach to smoothing in HMMs

Assuming that $\Pi_t(i) = p(S_t = i) > 0$ for all i and t, derive a recursive algorithm for updating $r_t(i) = p(S_t = i|\mathbf{x}_{t+1:T})$. Hint: it should be very similar to the standard forwards algorithm, but using a time-reversed transition matrix. Then show how to compute the posterior marginals $\gamma_t(i) = p(S_t = i|\mathbf{x}_{1:T})$

from the backwards filtered messages $r_t(i)$, the forwards filtered messages $\alpha_t(i)$, and the stationary distribution $\Pi_t(i)$.

Exercise 17.3 EM for for HMMs with mixture of Gaussian observations

Consider an HMM where the observation model has the form

$$p(\mathbf{x}_t|z_t = j, \boldsymbol{\theta}) = \sum_k w_{jk} \mathcal{N}(\mathbf{x}_t|\mu_{jk}, \boldsymbol{\Sigma}_{jk}) \tag{17.125}$$

- Draw the DGM.
- Derive the E step.
- Derive the M step.

Exercise 17.4 EM for for HMMs with tied mixtures

In many applications, it is common that the observations are high-dimensional vectors (e.g., in speech recognition, $\mathbf{x}_t$ is often a vector of cepstral coefficients and their derivatives, so $\mathbf{x}_t \in \mathbb{R}^{39}$), so estimating a full covariance matrix for KM values (where M is the number of mixture components per hidden state), as in Exercise 17.3, requires a lot of data. An alternative is to use just M Gaussians, rather than MK Gaussians, and to let the state influence the mixing weights but not the means and covariances. This is called a **semi-continuous HMM** or **tied-mixture HMM**.

- Draw the corresponding graphical model.
- Derive the E step.
- Derive the M step.