3

Markov chains and hidden Markov models

Having introduced some methods for pairwise alignment in Chapter 2, the emphasis will switch in this chapter to questions about a single sequence. The main aim of the chapter is to develop the theory for a very general form of probabilistic model for sequences of symbols, called a hidden Markov model (abbreviated HMM). The types of question we can use HMMs and their simpler cousins, Markov models, to consider are: 'Does this sequence belong to a particular family?' or 'Assuming the sequence does come from some family, what can we say about its internal structure?' An example of the second type of problem would be to try to identify alpha helix or beta sheet regions in a protein sequence.

As well as giving examples from the biological sequence world, we also give the mathematics and algorithms for many of the operations on HMMs in a more general form. These methods, or close analogues of them, are applied in many other sections of the book. This chapter therefore contains a fairly large amount of mathematically technical material. We have tried to organise it so that the first half, approximately, leads the reader through the essential algorithms using a single biological example. In the later sections we introduce a variety of other examples to illustrate more complex extensions of the basic approaches.

In the next chapter, we will see how HMMs can also be applied to the types of alignment problem discussed in Chapter 2, in Chapter 5 they are applied to searching databases for protein families, and in Chapter 6 to alignment of several sequences simultaneously. In fact, the search and alignment applications constitute probably the best-known use of HMMs for biological sequence analysis. However, we present HMM theory here in a less specialised context in order to emphasise its much broader applicability, which goes far beyond that of sequence alignment.

The overwhelming majority of papers on HMMs belong to the speech recognition literature, where they were applied first in the early 1970s. One of the best general introductions to the subject is the review by Rabiner [1989], which also covers the history of the topic. Although there will be quite a bit of overlap between that and the present chapter, there will be important differences in focus.

RabinerJuang93 added.

Before going on to introduce HMMs for biological sequence analysis, it is perhaps interesting to look briefly at how they are used for speech recognition [Rabiner & Juang 1993]. After recording, a speech signal is divided into pieces (called frames) of 10–20 milliseconds. After some preprocessing each frame is assigned to one out of a large number of predefined categories by a process known as vector quantisation. Typically there are 256 such categories. The speech signal is then represented as a long sequence of category labels and from that the speech recogniser has to find out what sequence of phonemes (or words) was spoken. The problems are that there are variations in the actual sound uttered, and there are also variations in the time taken to say the various parts of the word.

Many problems in biological sequence analysis have the same structure: based on a sequence of symbols from some alphabet, find out what the sequence represents. For proteins the sequences consist of symbols from the alphabet of 20 amino acids, and we typically want to know what protein family a given sequence belongs to. Here the primary sequence of amino acids is analogous to the speech signal and the protein family to the spoken word it represents. The time-variation of the speech signal corresponds to having insertions and deletions in the protein sequences.

Let us turn to a simpler example, which we will use to introduce first standard Markov models, of the non-hidden variety, then a simple hidden Markov model.

Example: CpG islands

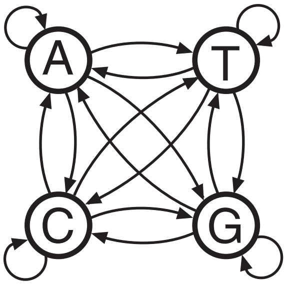
AK: A, C, G and T in tt font.

In the human genome wherever the dinucleotide `CG` occurs (frequently written `CpG` to distinguish it from the `C`-`G` base pair across the two strands) the `C` nucleotide (cytosine) is typically chemically modified by methylation. There is a relatively high chance of this methyl-`C` mutating into a `T`, with the consequence that in general `CpG` dinucleotides are rarer in the genome than would be expected from the independent probabilities of `C` and `G`. For biologically important reasons the methylation process is suppressed in short stretches of the genome, such as around the promoters or ‘start’ regions of many genes. In these regions we see many more `CpG` dinucleotides than elsewhere, and in fact more `C` and `G` nucleotides in general. Such regions are called CpG islands [Bird 1987]. They are typically a few hundred to a few thousand bases long.

We will consider two questions: Given a short stretch of genomic sequence, how would we decide if it comes from a CpG island or not? Second, given a long piece of sequence, how would we find the CpG islands in it, if there are any? Let us start with the first question. □

3.1 Markov chains

What sort of probabilistic model might we use for CpG island regions? We know that dinucleotides are important. We therefore want a model that generates sequences in which the probability of a symbol depends on the previous symbol. The simplest such model is a classical Markov chain. We like to show a Markov chain graphically as a collection of 'states', each of which corresponds to a particular residue, with arrows between the states. A Markov chain for DNA can be drawn like this:



where we see a state for each of the four letters A, C, G, and T in the DNA alphabet. A probability parameter is associated with each arrow in the figure, which determines the probability of a certain residue following another residue, or one state following another state. These probability parameters are called the *transition probabilities*, which we will write a_{st}:

AK: A, C, G and T in tt font.

$$a_{st} = P(x_i = t | x_{i-1} = s). \tag{3.1}$$

For any probabilistic model of sequences we can write the probability of the sequence as

$$\begin{aligned} P(x) &= P(x_L, x_{L-1}, \ldots, x_1) \\ &= P(x_L | x_{L-1}, \ldots, x_1) P(x_{L-1} | x_{L-2}, \ldots, x_1) \cdots P(x_1) \end{aligned}$$

by applying $P(X,Y) = P(X|Y)P(Y)$ many times. The key property of a Markov chain is that the probability of each symbol x_i depends only on the value of the preceding symbol x_{i-1}, not on the entire previous sequence, i.e. $P(x_i|x_{i-1}, \ldots, x_1)$ $= P(x_i|x_{i-1}) = a_{x_{i-1}x_i}$. The previous equation therefore becomes

$$\begin{aligned} P(x) &= P(x_L|x_{L-1}) P(x_{L-1}|x_{L-2}) \cdots P(x_2|x_1) P(x_1) \\ &= P(x_1) \prod_{i=2}^{L} a_{x_{i-1}x_i}. \end{aligned} \tag{3.2}$$

Although we have derived this equation in the context of CpG islands in DNA sequences, it is in fact the general equation for the probability of a specific sequence from any Markov chain. There is a large literature on Markov chains, see for example Cox & Miller [1965].

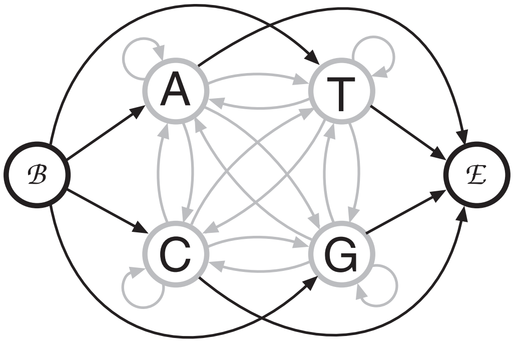


Figure 3.1 *Begin and end states can be added to a Markov chain (grey model) for modelling both ends of a sequence.*

Exercise

3.1 The sum of the probabilities of all possible sequences of length L can be written (using (3.2))

$$\sum_{\{x\}} P(x) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_L} P(x_1) \prod_{i=2}^{L} a_{x_{i-1}x_i}.$$

Show that this is equal to 1.

Modelling the beginning and end of sequences

Notice that as well as specifying the transition probabilities we must also give the probability $P(x_1)$ of starting in a particular state. To avoid the inhomogeneity of (3.2) introduced by the starting probabilities, it is possible to add an extra *begin state* to the model. At the same time we add a letter to the alphabet, which we will call $\mathcal{B}$. By defining $x_0 = \mathcal{B}$ the beginning of a sequence is also included in (3.2), so for instance the probability of the first letter in the sequence is

$$P(x_1 = s) = a_{\mathcal{B}s}.$$

Similarly we can add a symbol $\mathcal{E}$ to the end of a sequence to ensure the end is modelled. Then the probability of ending with residue t is

$$P(\mathcal{E}|x_L = t) = a_{t\mathcal{E}}.$$

To match the new symbols, we add begin and end states to the DNA model (see Figure 3.1). In fact, we need not explicitly add any letters to the alphabet, but instead can treat the two new states as ‘silent’ states that just serve as start and end points.

Traditionally the end of a sequence is not modelled in Markov chains; it is assumed that the sequence can end anywhere. The effect of adding an explicit end state is to model a distribution of lengths of the sequence. This way the model defines a probability distribution over all possible sequences (of any length). The distribution over lengths decays exponentially; see the exercise below.

Exercises

3.2 Assume that the model has an end state, and that the transition from any state to the end state has probability τ. Show that the sum of the probabilities (3.2) over all sequences of length L (and properly terminating by making a transition to the end state) is $\tau(1-\tau)^{L-1}$.

3.3 Show that the sum of the probability over all possible sequences of any length is 1. This proves that the Markov chain really describes a proper probability distribution over the whole space of sequences. (Hint: Use the result that, for $0 < x < 1$, $\sum_{i=0}^{\infty} x^i = 1/(1-x)$.)

Using Markov chains for discrimination

A primary use for equation (3.2) is to calculate the values for a likelihood ratio test. We illustrate this here using real data for the CpG island example. From a set of human DNA sequences we extracted a total of 48 putative CpG islands and derived two Markov chain models, one for the regions labelled as CpG islands (the '+' model) and the other from the remainder of the sequence (the '−' model). The transition probabilities for each model were set using the equation

$$a_{st}^{+} = \frac{c_{st}^{+}}{\sum_{t'} c_{st'}^{+}}, \tag{3.3}$$

and its analogue for a_{st}^{-}, where c_{st}^{+} is the number of times letter t followed letter s in the labelled regions. These are the maximum likelihood (ML) estimators for the transition probabilities, as described in Chapter 1.

(In this case there were almost 60 000 nucleotides, and ML estimators are adequate. If the number of counts of each type had been small, then a Bayesian estimation process would have been more appropriate, as discussed in Chapter 11 and below for HMMs.) The resulting tables are

'for below HMMs' changed to 'below for HMMs'.

+	A	C	G	T
A	0.180	0.274	0.426	0.120
C	0.171	0.368	0.274	0.188
G	0.161	0.339	0.375	0.125
T	0.079	0.355	0.384	0.182

−	A	C	G	T
A	0.300	0.205	0.285	0.210
C	0.322	0.298	0.078	0.302
G	0.248	0.246	0.298	0.208
T	0.177	0.239	0.292	0.292

where the first row in each case contains the frequencies with which an A is followed by each of the four bases, and so on for the other rows, so each row sums to one. These numbers are not the same; for example, G following A is much more common than T following A. Notice also that the tables are asymmetric. In both tables the probability for G following C is lower than that for C following G, although the effect is stronger in the '$-$' table, as expected.

To use these models for discrimination, we calculate the log-odds ratio

$$\begin{aligned} S(x) &= \log \frac{P(x|\text{model }+)}{P(x|\text{model }-)} = \sum_{i=1}^{L} \log \frac{a^{+}_{x_{i-1}x_i}}{a^{-}_{x_{i-1}x_i}} \\ &= \sum_{i=1}^{L} \beta_{x_{i-1}x_i} \end{aligned}$$

where x is the sequence and $\beta_{x_{i-1}x_i}$ are the log likelihood ratios of corresponding transition probabilities. A table for β is given below in bits:[1]

β	A	C	G	T
A	−0.740	0.419	0.580	−0.803
C	−0.913	0.302	1.812	−0.685
G	−0.624	0.461	0.331	−0.730
T	−1.169	0.573	0.393	−0.679

Figure 3.2 shows the distribution of scores, $S(x)$, normalised by dividing by their length, i.e. as an average number of bits per molecule. If we had not normalised by length, the distribution would have been much more spread out.

We see a reasonable discrimination between regions labelled CpG island and other regions. The discrimination is not very much influenced by the length normalisation. If we wanted to pursue this further and investigate the cases of misclassification, it is worth remembering that the error could either be due to an inadequate or incorrectly parameterised model, or to mislabelling of the training data.

3.2 Hidden Markov models

There are a number of extensions to classical Markov chains, which we will come back to later in the chapter. Here, however, we will proceed immediately to hidden Markov models. We will motivate this by turning to the second of the two questions posed initially for CpG islands: How do we find them in a long unannotated sequence? The Markov chain models that we have just built could be used for this purpose, by calculating the log-odds score for a window of, say, 100 nucleotides around every nucleotide in the sequence and plotting it. We would then

[1] Base 2 logarithms were used, in which case the unit is called a bit. See Chapter 11.

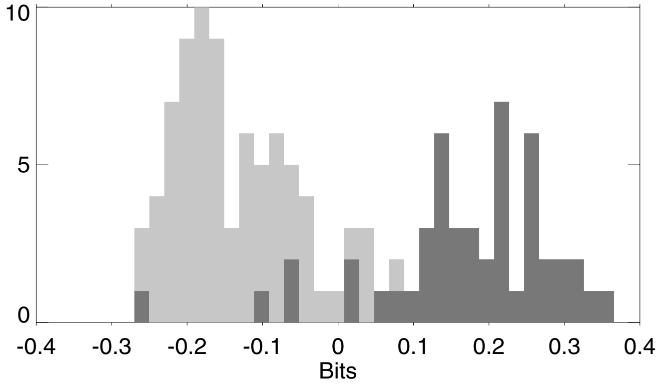


Figure 3.2 *The histogram of the length-normalised scores for all the sequences. CpG islands are shown with dark grey and non-CpG with light grey.*

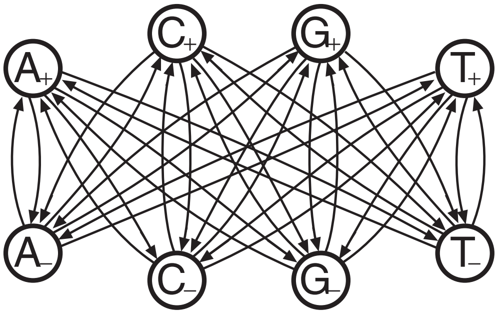


Figure 3.3 *An HMM for CpG islands. In addition to the transitions shown, there is also a complete set of transitions within each set, as in the earlier simple Markov chains.*

expect CpG islands to stand out with positive values. However, this is somewhat unsatisfactory if we believe that in fact CpG islands have sharp boundaries, and are of variable length. Why use a window size of 100? A more satisfactory approach is to build a single model for the entire sequence that incorporates both Markov chains.

To simulate in one model the ‘islands’ in a ‘sea’ of non-island genomic sequence, we want to have both the Markov chains of the last section present in the same model, with a small probability of switching from one chain to the other at each transition point. However, this introduces the complication that we now have two states corresponding to each nucleotide symbol. We resolve this by relabelling the states. We now have A_+, C_+, G_+ and T_+ which emit `A`, `C`, `G` and `T` respectively in CpG island regions, and A_-, C_-, G_- and T_- correspondingly in non-island regions; see Figure 3.3.

The transition probabilities in this model are set so that within each group they are close to the transition probabilities of the original component model, but there is also a small but finite chance of switching into the other component. Overall there is more chance of switching from '$+$' to '$-$' than vice versa, so if left to run free, the model will spend more of its time in the '$-$' non-island states than in the island states.

The relabelling is the critical step. The essential difference between a Markov chain and a hidden Markov model is that for a hidden Markov model there is not a one-to-one correspondence between the states and the symbols. It is no longer possible to tell what state the model was in when x_i was generated just by looking at x_i. In our example there is no way to tell by looking at a single symbol C in isolation whether it was emitted by state C_+ or state C_-

Formal definition of an HMM

Let us formalise the notation for hidden Markov models, and derive the probability of a particular sequence of states and symbols. We now need to distinguish the sequence of states from the sequence of symbols. Let us call the state sequence the *path*, π. The path itself follows a simple Markov chain, so the probability of a state depends only on the previous state. The ith state in the path is called π_i. The chain is characterised by parameters

$$a_{kl} = P(\pi_i = l | \pi_{i-1} = k). \tag{3.4}$$

To model the beginning of the process we introduce a begin state, as was introduced earlier to model the beginning of sequences in Markov chains (Figure 3.1). The transition probability a_{0k} from this begin state to state k can be thought of as the probability of starting in state k. It is also possible to model ends as before by always ending a state sequence with a transition into an end state. For convenience we label both begin and end states as 0 (there is no conflict here because you can only transit out of the begin state, and only into the end state, so variables are not used more than once).

Because we have decoupled the symbols b from the states k, we must introduce a new set of parameters for the model, $e_k(b)$. For our CpG model each state is associated with a single symbol, but this is not a requirement; in general a state can produce a symbol from a distribution over all possible symbols. We therefore define

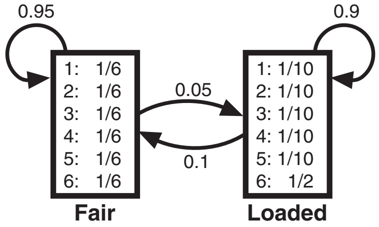
$$e_k(b) = P(x_i = b | \pi_i = k), \tag{3.5}$$

the probability that symbol b is seen when in state k. These are known as the *emission* probabilities.

For our CpG island model the emission probabilities are all 0 or 1. To illustrate emission probabilities we reintroduce here the casino example from Chapter 1.

Example: The occasionally dishonest casino, part 1

Let us consider an example from Chapter 1. In a casino they use a fair die most of the time, but occasionally they switch to a loaded die. The loaded die has probability 0.5 of a six and probability 0.1 for the numbers one to five. Assume that the casino switches from a fair to a loaded die with probability 0.05 before each roll, and that the probability of switching back is 0.1. Then the switch between dice is a Markov process. In each state of the Markov process the outcomes of a roll have different probabilities, and thus the whole processs is an example of a hidden Markov model. We can draw it like this:



where the emission probabilities $e()$ are shown in the state boxes. □

What is hidden in the above model? If you can just see a sequence of rolls (the sequence of observations) you do not know which rolls used a loaded die and which used a fair one, because that is kept secret by the casino; that is, the *state sequence is hidden*. In a Markov chain you always know exactly in which state a given observation belongs. Obviously the casino wouldn't tell you that they use loaded dice and what the various probabilities are. Yet for this more complicated situation, which we will return to later, it is possible to estimate the probabilities in the above HMM (once you have a suspicion that they use two different dice).

The reason for the name *emission* probabilities is that it is often convenient to think of HMMs as generative models, that generate or emit sequences. For instance we can generate random sequences of rolls from the model of the fair/-loaded dice above by simulating the successive choices of die, then rolls of the chosen die. More generally a sequence can be generated from an HMM as follows: First a state π_1 is chosen according to the probabilities a_{0i}. In that state an observation is emitted according to the distribution e_{π_1} for that state. Then a new state π_2 is chosen according to the transition probabilities $a_{\pi_1 i}$ and so forth. This way a sequence of random, artificial observations are generated. Therefore, we will sometimes say things like $P(x)$ is the probability that x was *generated* by the model.

It is now easy to write down the joint probability of an observed sequence x and a state sequence π:

$$P(x,\pi) = a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) a_{\pi_i \pi_{i+1}}, \tag{3.6}$$

where we require $\pi_{L+1} = 0$. For example, the probability of sequence CGCG being emitted by the state sequence $(\mathrm{C}_+, \mathrm{G}_-, \mathrm{C}_-, \mathrm{G}_+)$ in our model is

$$a_{0,\mathrm{C}_+} \times 1 \times a_{\mathrm{C}_+,\mathrm{G}_-} \times 1 \times a_{\mathrm{G}_-,\mathrm{C}_-} \times 1 \times a_{\mathrm{C}_-,\mathrm{G}_+} \times 1 \times a_{\mathrm{G}_+,0}.$$

Equation (3.6) is the HMM analogue of equation (3.2). However, it is not so useful in practice because in general we do not know the path. In the following sections we describe how to estimate the path, either by finding the most likely one, or alternatively by using an *a posteriori* distribution over states. Then we go on to show how to estimate the parameters for an HMM.

Most probable state path: the Viterbi algorithm

Although it is no longer possible to tell what state the system is in by looking at the corresponding symbol, it is often the sequence of underlying states that we are interested in. To find out what the observation sequence 'means' by considering the underlying states is called *decoding* in the jargon of speech recognition. There are several approaches to decoding. Here we will describe the most common one, called the Viterbi algorithm. It is a dynamic programming algorithm closely related to the ones covered in Chapter 2.

In general there may now be many state sequences that could give rise to any particular sequence of symbols. For example, in our CpG model the state sequences $(\mathrm{C}_+, \mathrm{G}_+, \mathrm{C}_+, \mathrm{G}_+)$, $(\mathrm{C}_-, \mathrm{G}_-, \mathrm{C}_-, \mathrm{G}_-)$ and $(\mathrm{C}_+, \mathrm{G}_-, \mathrm{C}_+, \mathrm{G}_-)$ would all generate the symbol sequence CGCG. However, they do so with very different probabilities. The third is the product of multiple small probabilities of switching back and forth between the components, and hence is much smaller than the first two. The second is itself significantly smaller than the first because it contains two C to G transitions which are significantly less probable in the '$-$' component than in the '$+$' component. Of these three choices, therefore, it is most likely that the sequence CGCG came from a set of '$+$' states.

A predicted path through the HMM will tell us which part of the sequence is predicted as a CpG island, because we assumed above that each state was assigned to model either CpG islands or other regions. If we are to choose just one path for our prediction, perhaps the one with the highest probability should be chosen,

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, P(x, \pi). \tag{3.7}$$

The most probable path π^* can be found recursively. Suppose the probability $v_k(i)$ of the most probable path ending in state k with observation i is known for all the states k. Then these probabilities can be calculated for observation x_{i+1} as

$$v_l(i+1) = e_l(x_{i+1}) \max_k (v_k(i) a_{kl}). \tag{3.8}$$

v		C	G	C	G
$\mathcal{B}$	1	0	0	0	0
A_+	0	0	0	0	0
C_+	0	**0.13**	0	**0.012**	0
G_+	0	0	**0.034**	0	**0.0032**
T_+	0	0	0	0	0
A_-	0	0	0	0	0
C_-	0	0.13	0	0.0026	0
G_-	0	0	0.010	0	0.00021
T_-	0	0	0	0	0

Figure 3.4 *For the model of CpG islands shown in Figure 3.3 and the sequence* CGCG*, this is the resulting table of v. The most probable path is shown with bold face.*

All sequences have to start in state 0 (the begin state), so the initial condition is that $v_0(0) = 1$. By keeping pointers backwards, the actual state sequence can be found by backtracking. The full algorithm is:

Algorithm: Viterbi

Initialisation ($i = 0$): $v_0(0) = 1$, $v_k(0) = 0$ for $k > 0$.

Recursion ($i = 1 \ldots L$): $v_l(i) = e_l(x_i)\max_k(v_k(i-1)a_{kl})$;
$\mathrm{ptr}_i(l) = \mathrm{argmax}_k(v_k(i-1)a_{kl})$.

Termination: $P(x, \pi^*) = \max_k(v_k(L)a_{k0})$;
$\pi^*_L = \mathrm{argmax}_k(v_k(L)a_{k0})$.

Traceback ($i = L \ldots 1$): $\pi^*_{i-1} = \mathrm{ptr}_i(\pi^*_i)$. ◁

Note that an end state is assumed, which is the reason for a_{k0} in the termination step. If ends are not modelled, this a will disappear.

There are some implementational issues both for the Viterbi algorithm and the algorithms described later. The most severe practical problem is that multiplying many probabilities always yields very small numbers that will give underflow errors on any computer. For this reason the Viterbi algorithm should always be done in log space, i.e. calculating $\log(v_l(i))$, which will make the products become sums and the numbers stay reasonable. This is discussed in Section 3.6.

Figure 3.4 shows the full table of values of v for the sequence CGCG and the CpG island model. When we apply the same algorithm to a longer sequence the derived optimal path π^* will switch between the '$+$' and the '$-$' components of the model, and thereby give the precise boundaries of the predicted CpG island regions.

Example: The occasionally dishonest casino, part 2

For a sequence of dice rolls we can now find the most probable path through the model shown on p. 55. A total of 300 random rolls were generated from the

Example reference by page number

```
Rolls   315116246446644245311321631164152133625144543631656626566666
Die     FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLL
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLL

Rolls   651166453132651245636664631636663162326455236266666625151631
Die     LLLLLLFFFFFFFFFFFFLLLLLLLLLLLLLLLLFFFLLLLLLLLLLLLLLLFFFFFFFFF
Viterbi LLLLLLFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLFFFFFFFF

Rolls   222555441666566563564324364131513465146353411126414626253356
Die     FFFFFFFFLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLL
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFL

Rolls   366163666466232534413661661163252562462255265252266435353336
Die     LLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Viterbi LLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Rolls   233121625364414432335163243633665562466662632666612355245242
Die     FFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLLFFFFFFFFFFF
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLFFFFFFFFFFF
```

Figure 3.5 *The numbers show 300 rolls of a die as described in the example. Below is shown which die was actually used for that roll (F for fair and L for loaded). Under that the prediction by the Viterbi algorithm is shown.*

model as described earlier. Each roll was generated either with the fair die (F) or the loaded one (L), as shown below the outcome of the roll in Figure 3.5. The Viterbi algorithm was used to predict the state sequence, i.e. which die was used for each of the rolls. Generally, as you can see, the Viterbi algorithm has recovered the state sequence fairly well. □

π moved to below argmax.

Exercise

3.4 Show that $\pi^* = \operatorname*{argmax}_{\pi} P(\pi|x)$ is equivalent to (3.7).

The forward algorithm

For Markov chains we calculated the probability of a sequence, $P(x)$, with equation (3.2). The resulting values were used to distinguish between CpG islands and other DNA for instance. We want to be able to calculate this probability for an HMM as well. Because many different state paths can give rise to the same sequence x, we must add the probabilities for all possible paths to obtain the full probability of x,

$$P(x) = \sum_{\pi} P(x, \pi). \tag{3.9}$$

The number of possible paths π increases exponentially with the length of the sequence, so brute force evaluation of (3.9) by enumerating all paths is not practical. One approach is to use equation (3.6) evaluated at the most probable path π^* obtained in the last section as an approximation to $P(x)$. This implicitly assumes that the only path with significant probability is π^*, a somewhat startling assumption which however in many cases is surprisingly good. In fact the approximation is unnecessary, because the full probability can itself be calculated by a similar dynamic programming procedure to the Viterbi algorithm, replacing the maximisation steps with sums. This is called the *forward* algorithm.

The quantity corresponding to the Viterbi variable $v_k(i)$ in the forward algorithm is

$$f_k(i) = P(x_1 \dots x_i, \pi_i = k), \tag{3.10}$$

which is the probability of the observed sequence up to and including x_i, requiring that $\pi_i = k$. The recursion equation is

$$f_l(i+1) = e_l(x_{i+1}) \sum_k f_k(i) a_{kl}. \tag{3.11}$$

The full algorithm is:

Algorithm: Forward algorithm

Initialisation ($i = 0$): $f_0(0) = 1,\ f_k(0) = 0$ for $k > 0$.

Recursion ($i = 1 \dots L$): $f_l(i) = e_l(x_i) \sum_k f_k(i-1) a_{kl}$.

Termination: $P(x) = \sum_k f_k(L) a_{k0}$. ◁

Like the Viterbi algorithm, the forward algorithm (and the backward algorithm in the next section) can give underflow errors when implemented on a computer. Again this can be solved by working in log space, although not as elegantly as for Viterbi. Alternatively a scaling method can be used. Both approaches are described in Section 3.6.

As well as their use in the forward algorithm, the quantities $f_k(i)$ have a number of other uses, including those described in the next two sections.

The backward algorithm and posterior state probabilities

The Viterbi algorithm finds the most probable path through the model, but as we remarked at the time, this may not always be the most appropriate basis for further inference about the sequence. We might for instance want to know what the most probable state is for an observation x_i. More generally, we may want the probability that observation x_i came from state k given the observed sequence, i.e. $P(\pi_i = k|x)$. This is the posterior probability of state k at time i when the emitted sequence is known.

Our approach to the posterior probability is a little indirect. We first calculate the probability of producing the entire observed sequence with the ith symbol being produced by state k:

$$\begin{aligned} P(x,\pi_i = k) &= P(x_1 \ldots x_i, \pi_i = k) P(x_{i+1} \ldots x_L | x_1 \ldots x_i, \pi_i = k) \\ &= P(x_1 \ldots x_i, \pi_i = k) P(x_{i+1} \ldots x_L | \pi_i = k), \end{aligned} \tag{3.12}$$

the second row following because everything after k only depends on the state at k. The first term in this is recognised as $f_k(i)$ from (3.10) that was calculated by the forward algorithm of the previous section. The second term is called $b_k(i)$,

$$b_k(i) = P(x_{i+1} \ldots x_L | \pi_i = k). \tag{3.13}$$

It is analogous to the forward variable, but instead obtained by a backward recursion starting at the end of the sequence:

Algorithm: Backward algorithm

Initialisation ($i = L$): $b_k(L) = a_{k0}$ for all k.

Recursion ($i = L-1, \ldots, 1$): $b_k(i) = \sum_l a_{kl} e_l(x_{i+1}) b_l(i+1)$.

Termination: $P(x) = \sum_l a_{0l} e_l(x_1) b_l(1)$. ◁

The termination step is rarely needed, because $P(x)$ is usually found by the forward algorithm, and it is just shown for completeness.

Equation (3.12) can now be written as $P(x, \pi_i = k) = f_k(i) b_k(i)$, and from it we obtain the required posterior probabilities by straightforward conditioning,

$$P(\pi_i = k | x) = \frac{f_k(i) b_k(i)}{P(x)}, \tag{3.14}$$

where $P(x)$ is the result of the forward (or backward) calculation.

Example: The occasionally dishonest casino, part 3

In Figure 3.6 the posterior probability for the die being fair is shown for the sequence of rolls shown in Figure 3.5. Notice that the posterior probability does not reflect which die was actually used in some places. This is to be expected, simply because a misleading sequence of rolls can occur at random. □

Posterior decoding

A major use of the $P(\pi_i = k | x)$ is for two alternative forms of decoding in addition to the Viterbi decoding we introduced in the previous section. These are particularly useful when many different paths have almost the same probability as the most probable one, because then it is not well justified to consider only the most probable path.

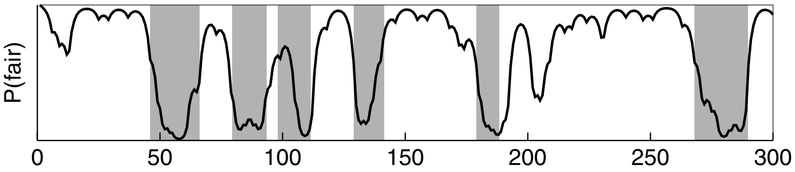


Figure 3.6 *The posterior probability of being in the state corresponding to the fair die in the casino example. The x axis shows the number of the roll. The shaded areas show when the roll was generated by the loaded die.*

The first approach is to define a state sequence $\hat{\pi}_i$ that can be used in place of π_i^*,

$$\hat{\pi}_i = \operatorname*{argmax}_k P(\pi_i = k|x). \tag{3.15}$$

As suggested by its definition, this state sequence may be more appropriate when we are interested in the state assignment at a particular point i, rather than the complete path. In fact, the state sequence defined by $\hat{\pi}_i$ may not be particularly likely as a path through the entire model; it may even not be a legitimate path at all if some transitions are not permitted, which is normally the case.

The second, and perhaps more important, new decoding approach arises when it is not the state sequence itself which is of interest, but some other property derived from it. Assume we have a function $g(k)$ defined on the states. The natural value to look at then is

$$G(i|x) = \sum_k P(\pi_i = k|x)g(k). \tag{3.16}$$

An important special case of this is where $g(k)$ takes the value 1 for a subset of the states and 0 for the rest. In this case, $G(i|x)$ is the posterior probability of the symbol i coming from a state in the specified set. For example, with our CpG island model, what really concerns us is whether a base is part of an island or not. For this purpose we want to define $g(k) = 1$ for $k \in \{\mathrm{A}_+, \mathrm{C}_+, \mathrm{G}_+, \mathrm{T}_+\}$ and $g(k) = 0$ for $k \in \{\mathrm{A}_-, \mathrm{C}_-, \mathrm{G}_-, \mathrm{T}_-\}$. Then $G(i|x)$ is precisely the posterior probability according to the model that base i is in a CpG island.

In the case where we have a labelling of the states defining a partition of them (as we in fact have with the CpG island model, labelling them as '$+$' or '$-$') it is possible to use (3.16) to find the most probable label at each position of the sequence. This is not quite the most probable global labelling of a given sequence. That, however, is not entirely straightforward. See Schwartz & Chow [1990] and Krogh [1997b] for further discussion of this. Change begin

Example: Prediction of CpG islands

Now CpG islands can be predicted from our model. By the Viterbi algorithm we can find the most probable path through the model. When this path goes through

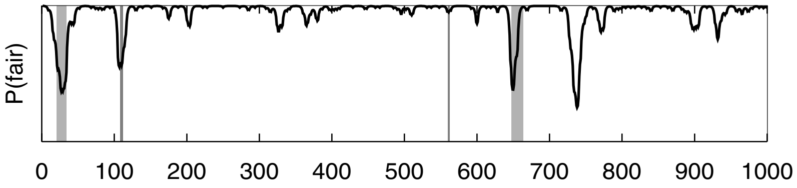


Figure 3.7 *The posterior probability of the die being fair, but using probability 0.01 for switching to the loaded die (cf. Figure 3.6).*

the + states, a CpG island is predicted. For the set of 41 sequences, each with a putative CpG island, all the islands are found except for two (false negatives), and 121 new ones are predicted (false positives). The real CpG islands are quite long (of the order of 1000 bases), whereas the predicted ones are short, and a CpG island is usually predicted as several short ones. By applying the two simple post-processing steps (1) concatenate predictions less than 500 bases apart (2) discard predictions shorter than 500, the number of false positives are reduced to 67.

Using posterior decoding, the same two CpG islands are missed and 236 false positives are predicted. Using the same post-processing as above this number is reduced to 83. For this problem, there is not a big difference between the two methods, except that the posterior decoding predicts even more very short islands. It is possible that some of the false positives are real CpG islands. The two false negatives are perhaps wrongly labelled, but it is also possible that a more sophisticated model is needed for capturing all the features of these signals. □

Change end

Example: The occasionally dishonest casino, part 4

The model for the casino is changed, so there is only a probability of 0.01 for switching from fair to loaded. Obviously the probability of staying with the fair die must then be 0.99, but all other probabilities are unchanged. From this model 1000 random rolls are generated. From these rolls the most probable path found by the Viterbi algorithm never visits the loaded die state. In Figure 3.7 the posterior probability for the dice being fair is shown for these rolls. Although not perfect, posterior decoding would predict something reasonably close to the truth. □

3.3 Parameter estimation for HMMs

Probably the most difficult problem faced when using HMMs is that of specifying the model in the first place. There are two parts to this: the design of the structure, i.e. what states there are and how they are connected, and the assignment of parameter values, the transition and emission probabilities a_{kl} and $e_k(b)$. In this section we will discuss the parameter estimation problem, for which there is a well-developed theory. In the next section we will consider model structure design, which is more of an art.

The framework in which we will be working is to assume that we have a set of example sequences of the type that we want the model to fit well, known as *training* sequences. Let these be $x^1,\dots,x^n$. We assume that they are independent, and thus that the joint probability of all the sequences given a particular assignment of parameters is the product of the probabilities of the individual sequences. In fact, we work in log space, and so with the log probability of the sequences,

$$l(x^1,\dots,x^n|\theta) = \log P(x^1,\dots,x^n|\theta) = \sum_{j=1}^{n} \log P(x^j|\theta), \tag{3.17}$$

where θ represents the entire current set of values of the parameters in the model (all the as and es). This is equal to the log likelihood of the model; see Chapter 11.

Estimation when the state sequence is known

Just as it was easier to write down the probability of a sequence when the path was known, so it is easier to estimate the probability parameters when the paths are known for all the examples. Frequently this is the case. An example would be if we were given a set of genomic sequences in which the CpG islands were already labelled, based on experimental data. Other examples would be for an HMM that predicted secondary structure, with training sequences obtained from the set of proteins with known structures, or for an HMM predicting genes from genomic sequences, where the transcript structure has been determined by cDNA sequencing.

When all the paths are known, we can count the number of times each particular transition or emission is used in the set of training sequences. Let these be A_{kl} and $E_k(b)$. Then, as shown in Chapter 11, the maximum likelihood estimators for a_{kl} and $e_k(b)$ are given by

$$a_{kl} = \frac{A_{kl}}{\sum_{l'} A_{kl'}} \quad \text{and} \quad e_k(b) = \frac{E_k(b)}{\sum_{b'} E_k(b')}. \tag{3.18}$$

The estimation equation for a_{kl} is exactly the same as for a simple Markov chain.

As always, maximum likelihood estimators are vulnerable to overfitting if there are insufficient data. Indeed if there is a state k that is never used in the set of example sequences, then the estimation equations are undefined for that state, because both the numerator and denominator will have value zero. To avoid such problems it is preferable to add predetermined pseudocounts to the A_{kl} and $E_k(b)$ before using (3.18).

$$\begin{aligned} A_{kl} &= \text{number of transitions } k \text{ to } l \text{ in training data} + r_{kl}, \\ E_k(b) &= \text{number of emissions of } b \text{ from } k \text{ in training data} + r_k(b). \end{aligned}$$

The pseudocounts r_{kl} and $r_k(b)$ should reflect our prior biases about the probability values. In fact they have a natural probabilistic interpretation as the parameters of Bayesian Dirichlet prior distributions on the probabilities for each state (see Chapter 11). They must be positive, but do not need to be integers. Small total values $\sum_{l'} r_{kl'}$ or $\sum_{b'} r_k(b')$ indicate weak prior knowledge, whereas larger total values indicate more definite prior knowledge, which requires more data to modify it.

Estimation when paths are unknown: Baum–Welch and Viterbi training

When the paths are unknown for the training sequences, there is no longer a direct closed-form equation for the estimated parameter values, and some form of iterative procedure must be used. All the standard algorithms for optimisation of continuous functions can be used; see for example Press *et al.* [1992]. However, there is a particular iteration method that is standardly used, known as the Baum–Welch algorithm [Baum 1972]. This has a natural probabilistic interpretation. Informally, it first estimates the A_{kl} and $E_k(b)$ by considering probable paths for the training sequences using the current values of a_{kl} and $e_k(b)$. Then (3.18) is used to derive new values of the as and es. This process is iterated until some stopping criterion is reached.

It is possible to show that the overall log likelihood of the model is increased by the iteration, and hence that the process will converge to a local maximum. Unfortunately, there are usually many local maxima, and which one you end up with depends strongly on the starting values of the parameters. The problem of local maxima is particularly severe when estimating large HMMs, and later we will discuss various ways to help deal with it.

More formally, the Baum–Welch algorithm calculates A_{kl} and $E_k(b)$ as the *expected* number of times each transition or emission is used, given the training sequences. To do this it uses the same forward and backward values as the posterior probability decoding method. The probability that a_{kl} is used at position i in sequence x is (see Exercise 3.5)

$$P(\pi_i = k, \pi_{i+1} = l | x, \theta) = \frac{f_k(i) a_{kl} e_l(x_{i+1}) b_l(i+1)}{P(x)}. \tag{3.19}$$

From this we can derive the expected number of times that a_{kl} is used by summing over all positions and over all training sequences,

$$A_{kl} = \sum_j \frac{1}{P(x^j)} \sum_i f_k^j(i) a_{kl} e_l(x_{i+1}^j) b_l^j(i+1), \tag{3.20}$$

where $f_k^j(i)$ is the forward variable $f_k(i)$ defined in (3.10) calculated for sequence

j, and $b_l^j(i)$ is the corresponding backward variable. Similarly, we can find the expected number of times that letter b appears in state k,

$$E_k(b) = \sum_j \frac{1}{P(x^j)} \sum_{\{i|x_i^j=b\}} f_k^j(i) b_k^j(i), \tag{3.21}$$

where the inner sum is only over those positions i for which the symbol emitted is b.

Having calculated these expectations, the new model parameters are calculated just as before using (3.18). We can iterate using the new values of the parameters to obtain new values of the As and Es as before, but in this case we are converging in a continuous-valued space, and so will never in fact reach the maximum. It is therefore necessary to set a convergence criterion, typically stopping when the change in total log likelihood is sufficiently small. Other stop criteria than the log likelihood change can be used for the iteration. For instance the log likelihood can be normalised by the number of sequences n and maybe also by the sequence lengths, so that you consider the change in the average log likelihood per residue. We can summarise the Baum–Welch algorithm like this:

Algorithm: Baum–Welch

```
Initialisation: Pick arbitrary model parameters.
Recurrence:
    Set all the $A$ and $E$ variables to their pseudocount values $r$ (or to zero).
    For each sequence $j = 1 \ldots n$:
        Calculate $f_k(i)$ for sequence $j$ using the forward algorithm (p. 59).
        Calculate $b_k(i)$ for sequence $j$ using the backward algorithm (p. 60).
        Add the contribution of sequence $j$ to $A$ (3.20) and $E$ (3.21).
    Calculate the new model parameters using (3.18).
    Calculate the new log likelihood of the model.
Termination:
    Stop if the change in log likelihood is less than some predefined threshold
    or the maximum number of iterations is exceeded.                          ◁
```

Page numbers added instead of algorithm numbers

As indicated here, it is normal to add pseudocounts to the A and E values just as in the case where the state paths are known. This works well, but the normal Bayesian interpretation in terms of Dirichlet priors does not carry through rigorously in this case; see Chapter 11.

The Baum–Welch algorithm is a special case of a very powerful general approach to probabilistic parameter estimation called the EM algorithm. This algorithm and the derivation of Baum–Welch is given in Section 11.6 of Chapter 11.

An alternative to the Baum–Welch algorithm is frequently used, which we will call Viterbi training. In this approach, the most probable paths for the training sequences are derived using the Viterbi algorithm given above, and these are used in the re-estimation process given in the previous section. Again, the process is iterated when the new parameter values are obtained. In this case the algorithm converges precisely, because the assignment of paths is a discrete process, and we can continue until none of the paths change. At this point the parameter estimates will not change either, because they are determined completely by the paths. Unlike Baum–Welch, this procedure does not maximise the true likelihood, i.e. $P(x^1,\ldots,x^n|\theta)$ regarded as a function of of the model parameters θ. Instead, it finds the value of θ that maximises the contribution to the likelihood $P(x^1,\ldots,x^n,\pi^*(x^1),\ldots,\pi^*(x^n)|\theta)$ from the most probable paths for all the sequences. Probably for this reason, Viterbi training performs less well in general than Baum–Welch. However, it is widely used, and it can be argued that when the primary use of the HMM is to produce decodings via Viterbi alignments, then it is good to train using them.

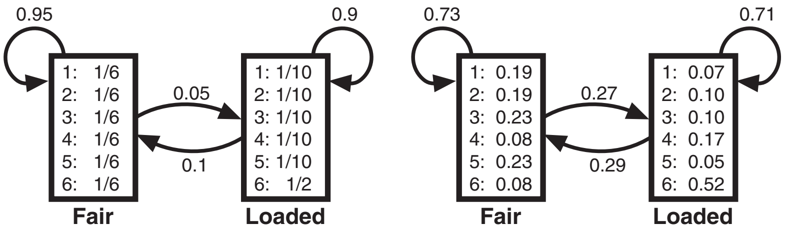
Change of wording to fit equation on line.

Example: The occasionally dishonest casino, part 5

We are suspicious that a casino is operated as described in the example on p. 55, but we do not know for certain. Night after night we collect data by simply observing rolls. When we have enough, we want to estimate a model. Assume the data we collected were the 300 rolls shown in Figure 3.5. From this sequence of observations a model was estimated by the Baum–Welch algorithm. Initially all the probabilities were set to random numbers. Here are diagrams of the model that generated the data (identical to the one in the example on p. 55) and the estimated model.

Numbered example reference changed.

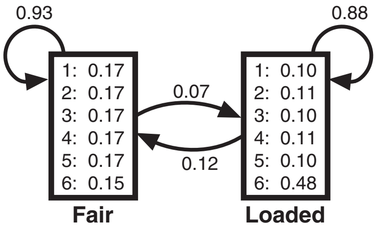
Numbered example reference changed.



You can see they are fairly similar, although the estimated transition probabilities are quite different from the real ones. This is partly a problem of local minima, and by trying more times it is actually possible to obtain a model closer to the correct one. However, from a limited amount of data it is never possible to estimate the parameters exactly.

To illustrate the last point, 30 000 random rolls were generated (data are not shown!), and a model was estimated. This came very close to the correct one:

‘data not shown’ changed to ‘data are not shown’ for nicer line break.



To see how good these models are compared to just assuming a fair die all the time, the log-odds per roll was calculated using the 300 observations for the three models:

The correct model	0.101 bits
Model estimated from 300 rolls	0.097 bits
Model estimated from 30 000 rolls	0.100 bits

The worst model estimated from 300 rolls has almost the same log-odds as the two other models. That is because it is being tested on the same data as it was estimated from. Testing it on an independent set of rolls yields significantly lower log-odds than the other two models. □

Exercises

3.5 Derive the result (3.19). Use the fact that

$$P(\pi_i = k, \pi_{i+1} = l|x,\theta) = \frac{1}{P(x|\theta)} P(x, \pi_i = k, \pi_{i+1} = l|\theta),$$

and that this again can be written in terms of $P(x_1,\ldots,x_i,\pi_i = k|\theta)$ and

$$\begin{aligned} P(x_{i+1},\ldots,x_L,\pi_{i+1} = l|x_1,\ldots,x_i,\theta,\pi_i = k) \\ = P(x_{i+1},\ldots,x_L,\pi_{i+1} = l|\theta,\pi_i = k). \end{aligned}$$

3.6 Derive (3.21).

Modelling of labelled sequences

In the above example with CpG islands we have seen how HMMs can be used to predict the labelling of unannotated sequences. In these examples we had to train the models of CpG islands separately from the model of non-CpG islands and then combine them into a larger HMM afterwards. This separate estimation can be quite tedious, especially if there are more than two different classes involved. Also, if the transitions between the submodels are ambiguous, so for instance a given sequence can use more than one transition from the CpG submodel to the other submodel, then the estimation of the transitions is not a simple counting

<table>
<tr><td colspan="3">Sequence</td><td>x_1</td><td>x_2</td><td>x_3</td><td>x_4</td><td>x_5</td><td>x_6</td><td>x_7</td><td>x_8</td><td>x_9</td><td>x_{10}</td><td>...</td></tr>
<tr><td colspan="3">Labels</td><td>$-$</td><td>$-$</td><td>$-$</td><td>$+$</td><td>$+$</td><td>$+$</td><td>$+$</td><td>$-$</td><td>$-$</td><td>$-$</td><td>...</td></tr>
<tr><td rowspan="8">State</td><td>1</td><td>$-$</td><td colspan="3" rowspan="4">f calculated as usual</td><td colspan="4" rowspan="4">$f=0$</td><td colspan="4" rowspan="4">f calculated as usual</td></tr>
<tr><td>2</td><td>$-$</td></tr>
<tr><td>3</td><td>$-$</td></tr>
<tr><td>4</td><td>$-$</td></tr>
<tr><td>5</td><td>$+$</td><td colspan="3" rowspan="4">$f=0$</td><td colspan="4" rowspan="4">f calculated as usual</td><td colspan="4" rowspan="4">$f=0$</td></tr>
<tr><td>6</td><td>$+$</td></tr>
<tr><td>7</td><td>$+$</td></tr>
<tr><td>8</td><td>$+$</td></tr>
</table>

Figure 3.8 *The forward table for a model with four states labelled $+$ and four labelled $-$. Each column corresponds to an observation and each row to a state of the model. The first ten residues shown, $x_1, \ldots, x_{10}$, are assumed to be labelled $---++++---$.*

problem. There is, however, a more straightforward method to estimate everything at once, which we will describe now.

The starting point is the combined model of all the classes, where we have assigned a class label to each state. To model CpG islands the natural labels are '$+$' for the island states and '$-$' for the non-island states. We also have labels on the observations $x = x_1, \ldots, x_L$, which we we call $y = y_1, \ldots, y_L$. The y_i is '$+$' if x_i is part of a CpG island and '$-$' otherwise. In the Baum–Welch algorithm (or the Viterbi alternative) we now only allow *valid* paths through the model when calculating the fs and bs. A valid path is one where the state labels and sequence labels are the same, i.e., π_i has label y_i. During the forward and backward algorithms this corresponds to setting $f_l(i) = 0$ and $b_l(i) = 0$ for all the states l with a label different from y_i (see Figure 3.8).

Discriminative estimation

Unless there are ambiguous transitions between submodels, the above estimation procedure gives the same result as if the submodels were estimated separately by the Baum–Welch algorithm and then combined with appropriate transitions afterwards. This actually corresponds to maximising the likelihood

$$\theta^{ML} = \underset{\theta}{\operatorname{argmax}} P(x, y|\theta).$$

'instead' added for good line break.

Usually our primary interest is in obtaining good predictions of y, so it is preferable to maximise $P(y|x,\theta)$ instead. This is called *conditional maximum likelihood* (CML),

$$\theta^{CML} = \underset{\theta}{\operatorname{argmax}} P(y|x,\theta); \qquad (3.22)$$

see for example Juang & Rabiner [1991] and Krogh [1994]. A related criterion is called *maximum mutual information* or MMI [Bahl *et al.* 1986].

The likelihood $P(y|x,\theta)$ can be rewritten as

$$P(y|x,\theta) = \frac{P(x,y|\theta)}{P(x|\theta)},$$

where $P(x,y|\theta)$ is the probability calculated by the forward algorithm for labelled sequences described above, and $P(x|\theta)$ is the probability calculated by the standard forward algorithm disregarding all the labels. There is no EM algorithm for optimising this likelihood, and the estimation becomes more complex; see for example Normandin & Morgera [1991] and the references above.

3.4 HMM model structure

Choice of model topology

So far we have assumed that transitions are possible from any state to any other state. Although it is tempting to start with a fully connected model, i.e. one in which all transitions are allowed, and 'let the model find out for itself' which transitions to use, it almost never works in practice. For problems of any realistic size it will usually lead to very bad models, even with plenty of training data. Here the problem is not over fitting, but local maxima. The less constrained the model is, the more severe the local maximum problem becomes. There are methods that attempt to adapt the model topology based on the data by adding and removing transitions and states [Stolcke & Omohundro 1993; Fujiwara, Asogawa & Konagaya 1994]. However, in practice successful HMMs are constructed by carefully deciding which transitions are to be allowed in the model, based on knowledge about the problem under investigation.

To disable the transition from state k to state l corresponds to setting $a_{kl} = 0$. If we use Baum–Welch estimation (or the Viterbi approximation) then a_{kl} will still be zero after the re-estimation process, because when the probability is zero the expected number of transitions from k to l will also be zero. Therefore all the mathematics is unchanged even if not all transitions are possible.

We should choose a model which has an interpretation in terms of our knowledge of the problem. For instance, to model CpG islands it was important that the model was capable of giving a different probability to a CG dinucleotide in the island states from in the non-island states, because that was expected to be the main determinator for CpG islands.

Duration modelling

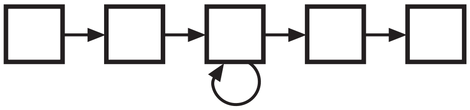
This section was moved to here from section 3.6

When modelling a phenomenon where for instance the nucleotide distribution does not change for a certain length of DNA, the simplest model design is to make a state with a transition to itself with probability p. We did this with both our CpG island and our dishonest casino example. After entering the state there *n* changed to *l*. Paren. added after eq.

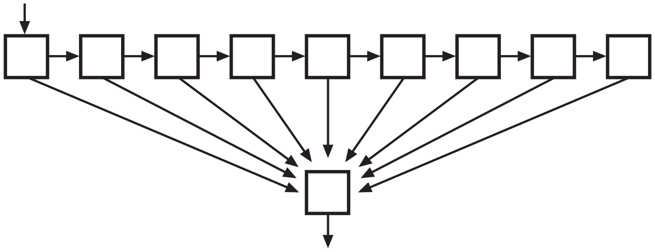
is a probability $1-p$ of leaving it, so the probability of staying in the state for l residues is

$$P(l \text{ residues}) = (1-p)p^{l-1}. \tag{3.23}$$

(The emission probabilities are disregarded.) This exponentially decaying distribution on lengths (called a geometric distribution) can be inappropriate in some applications, where the distribution of lengths is important and significantly different from exponential. More complex length distributions can be modelled by introducing several states with the same distribution over residues and transitions between each other. For instance a (sub-) model like this:



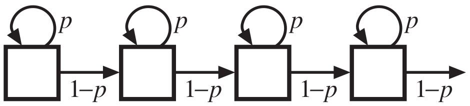
will give sequences of a minimum length of 5 residues and an exponentially decaying distribution over longer sequences. Similarly, a model like this:



Change begin

can model *any* distribution of lengths between 2 and 10.

A more subtle way of obtaining a non-geometric length distribution is to use an array of n states, each with a transition to itself of probability p and a transition to the next of probability $1-p$:



Obviously the smallest sequence length such a model can capture is n. For any given path of length l through the model, the probability of all its transitions is $p^{l-n}(1-p)^n$ (we are disregarding emission probabilities for now, as above). The number of possible paths through the states is $\binom{l-1}{n-1}$, so the total probability summed over all possible paths is

$$P(l) = \binom{l-1}{n-1} p^{l-n}(1-p)^n. \tag{3.24}$$

This distribution is called a negative binomial and it is shown in Figure 3.9 for $p = 0.99$ and $n \leq 5$. For small lengths the number of paths through the model grows faster than the geometrical distribution decays, and therefore the distribution becomes bell-shaped. The number of paths depends on the model topology, and it is possible to make more general models where the number of paths has a different dependence on n and l. For continuous Markov processes the types of

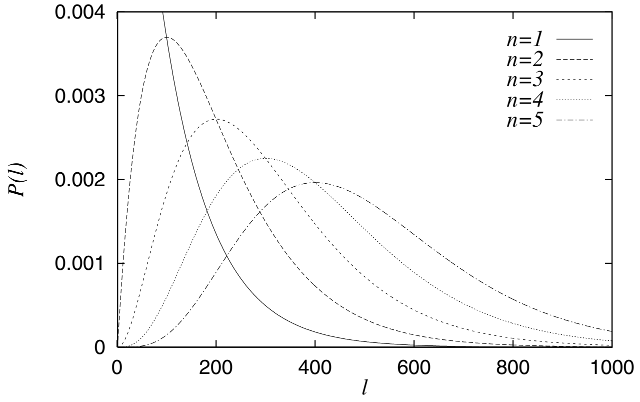


Figure 3.9 *The probability distribution over lengths for models with* $p = 0.99$ *and n identical states, with n ranging from* 1 *to* 5.

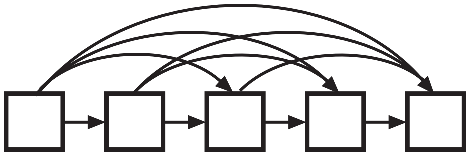
distributions that can be obtained are called Erlang distributions or more generally phase-type distributions, see for example Asmussen [1987]. Change end

Alternatively, it is possible to model the length distribution explicitly. As length is equivalent to time in many signal processing applications, this is called *duration modelling*. The price one has to pay is that algorithms are much slower. See Rabiner [1989] for more details.

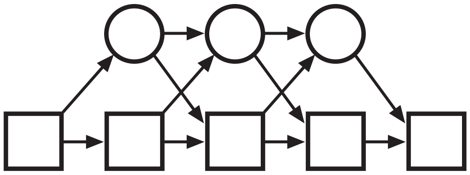
Silent states

We have already seen examples of states that do not emit symbols in an HMM, the begin and end states. Such states are called *silent states* or *null states*, and they can also be useful in other places in an HMM. In Chapter 5 we will see an example where all states in a chain of states need to be connected to all states later in the chain. The length of such a chain is often 200 states or more, and connecting them appropriately with transitions would require roughly 20 000 transition probabilities (assuming 200 states). This number is too large to be reliably estimated from realistic datasets. Instead, by using silent states, we can get away with around 800 transitions.

The situation is as follows: to allow for arbitrary deletions a chain of states needs to be completely 'forward connected'.



Instead we can connect all the states to a parallel chain of silent states, represented here by circles.



Because the silent states do not emit any letters, it is possible to get from any 'real' state to any later 'real' state without emitting any letters.

A price is paid for the reduction in the number of parameters. The fully connected model can have for instance high probability transitions from state 1 to state 5 and from state 2 to state 4, but low probability ones for transitions 1 to 4 and 2 to 5. This would not be possible with the model using silent states.

So long as there are no loops consisting entirely of silent states, it is easy to extend all the HMM algorithms to incorporate them. The condition that there are no loops mean that the states can be numbered so that any transition between silent states goes from a lower to a higher numbered state. For the forward algorithm, the change is as follows:

Change (i) For all 'real' states l, calculate $f_l(i+1)$ as before from $f_k(i)$ for states k.

Change (ii) For any silent state l, set $f_l(i+1)$ to $\sum_k f_k(i+1)a_{kl}$ for 'real' states k.

(iii) Starting from the lowest numbered silent state l add $\sum_k f_k(i+1)a_{kl}$ to $f_l(i+1)$ for all silent states $k < l$.

The change to the Viterbi algorithm is exactly the same (sums replaced by maximisation of course), and for the backward algorithm the change is essentially the same except in the third step the silent states are updated in reverse order.

If there are loops consisting entirely of silent states, the situation gets a little more complicated. It is possible to eliminate the silent states from the calculation by calculating (exactly) the effective transition probabilities between real states in the model, which involves inverting the transition matrix for the Markov model of silent states [Cox & Miller 1965]. Often, however, these effective transitions correspond to a fully connected model, and this leads to a substantial increase in the complexity of the model. Usually it is best to simply make sure such loops do not exist.

Exercises

3.7 Calculate the total number of transitions needed in a forward connected model as the one shown above with a length of L. Calculate the same number for a model with silent states (as above). Change begin

3.8 Show that the number of paths through an array of n states is indeed $\binom{l-1}{n-1}$ for length l as in (3.24).

3.9 Consider the model with n states with self-loops giving rise to equation (3.24). What is the probability for the most likely path through the model for a sequence of length l (when ignoring emission probabilities)? Is this type of length modelling useful with the Viterbi algorithm? Change end

3.5 More complex Markov chains

High order Markov chains

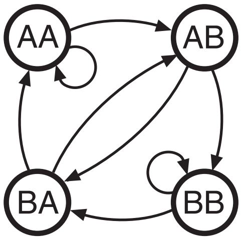
An nth order Markov process is a stochastic process where each event depends on the previous n events, so

$$P(x_i|x_{i-1},x_{i-2},\dots,x_1) = P(x_i|x_{i-1},\dots,x_{i-n}). \tag{3.25}$$

The Markov chains we have discussed so far are of order 1.

An nth order Markov chain over some alphabet $\mathcal{A}$ is equivalent to a first order Markov chain over the alphabet $\mathcal{A}^n$ of n-tuples. This follows from the simple fact that $P(x_k|x_{k-1}\dots x_{k-n}) = P(x_k,x_{k-1}\dots x_{k-n+1}|x_{k-1}\dots x_{k-n})$ (the probability of A and B given B is the probability of A given B). That is, the probability of x_k given the n-tuple ending in x_{k-1} is equal to the probability of the n-tuple ending in x_k given the n-tuple ending in x_{k-1}.

Consider the simple example of a second order Markov chain for sequences of only two different characters A and B. A sequence is translated to a sequence of pairs, so for instance the sequence ABBAB becomes AB-BB-BA-AB. The equivalent four-state first order Markov chain will look like this:



In this equivalent model not all transitions are allowed (or alternatively, some of the transition probabilities are zero). This is because only two different pairs can follow a given letter; the state AB for instance can only be followed by the states BA and BB. No sequence exists that can go from state AB to state AA. Similarly, a second order model for DNA is equivalent to a first order model over an alphabet of the 16 dinucleotides. A sequence of five bases, `CGTCA`, corresponds to a chain of four states, CG-GT-TC-CA, in a dinucleotide model.

Despite the theoretical equivalence between an nth order model and a first order model, the framework of high order models (meaning models of order greater Change begin
than 1) is sometimes more convenient. Theoretically the high order models are Change end
treated in a way completely equivalent to first order models.

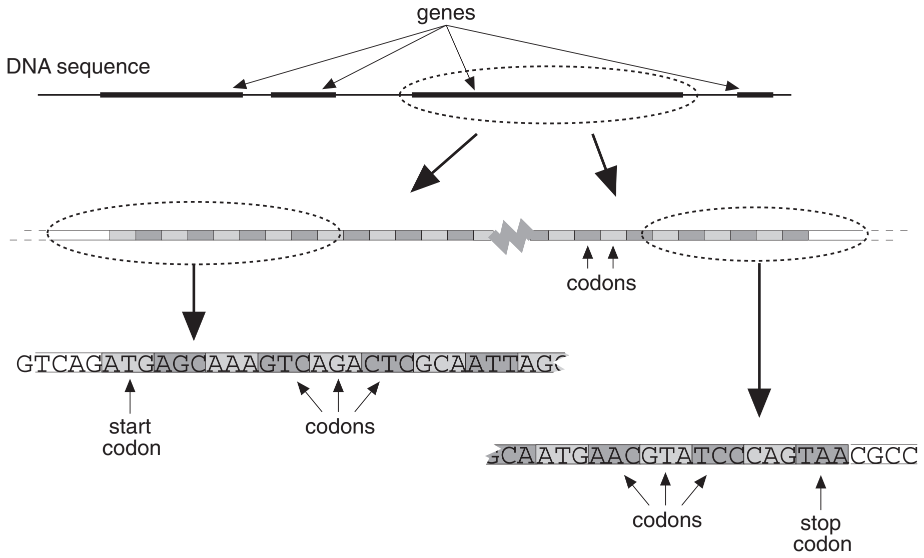


Figure 3.10 *The organisation of genes in prokaryotes.*

Finding prokaryotic genes

An example is given by a model for identifying prokaryotic genes. Genes of prokaryotes (bacteria) have a very simple one-dimensional structure. A gene coding for a protein starts with a start codon, then has a number of codons coding for amino acids, and ends with a stop codon; see Figure 3.10. Codons are DNA nucleotide triplets of which 61 code for amino acids and three are stop codons. In order to focus on the modelling, many complications such as frame shifts and non-protein genes are ignored here.

It is very easy to find good gene candidates by simply looking for stretches of DNA with the correct structure, i.e. starting with one of the three possible start codons, continuing with a number of non-stop codons and ending with one of the three stop codons. Such a gene candidate is called an *open reading frame* or just an ORF. Usually there are many overlapping ORFs that have the same stop codon, but different start codons. (The term ORF is often used for the maximal open reading frame between two stop codons, but we shall use it for all possible gene candidates.) There are many more ORFs than real genes, and here we will sketch possible ways of distinguishing between a non-coding ORF and a real gene.

In this example DNA from the bacterium *E. coli* is used (the dataset is described in detail in Krogh, Mian & Haussler [1994]). We consider only genes more than 100 nucleotides long. In the dataset there are 1100 such genes. This set is arbitrarily divided into a training set of 900 for training our models, and a test set containing the remaining 200 genes.

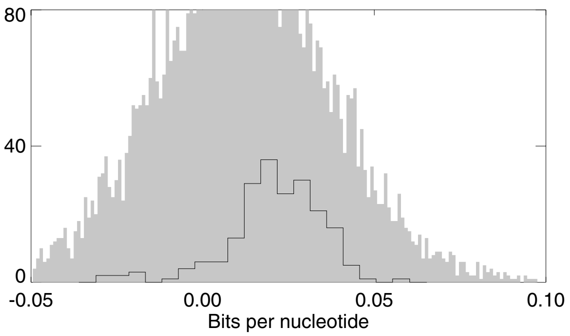


Figure 3.11 *Histograms of the log-odds per nucleotide for all NORFs (grey) and genes (black line) according to a first order Markov chain. Because of the large number of NORFs, the histogram bin size is five times smaller for the NORFs.*

We estimate a first order model just as we did for the CpG islands early in this chapter and test how well it discriminates genes from other ORFs. In the test set we found roughly 6500 ORFs with a length of more than 100 bases. ORFs that share the stop codon with a known real gene were not included, because they would generally score very well and make our subsequent analysis more difficult. The remaining ORFs that are not labelled as coding will be called NORFs (for non-coding ORFs).

In Figure 3.11 a histogram is shown of the log-odds per nucleotide. As the null model for calculating log-odds we used the simplest possible, with the probability for each nucleotide equal to the frequency by which it occurs in all the data. The average log-odds per nucleotide for all the genes is 0.018, whereas it is half as much (0.009) for the NORFs, but the variance makes it almost useless for discrimination. You could fool yourself into thinking that the model had a decent discriminative power if you plotted the histogram of log-odds without dividing by the sequence length, because the genes are longer on average than NORFS, and therefore also the total log-odds is larger for the NORFs. Almost all the apparent information about genes would come from the length distribution and not from the model.

It is worth noticing that the average of the histogram is *not* at 0 bits, and that the averages of the two distributions (genes and NORFs) are quite close. This indicates that the Markov chain has indeed found a non-random correlation between nucleotide pairs, but it is essentially the same in coding and non-coding regions. In a second order chain, the probability of a nucleotide depends on the two previous ones, so it spans the length of a codon. Therefore we also tried a second order model, but the result is almost identical to the one for the first order model, so we do not show the histogram. It would probably not help much to switch to a Markov chain of even higher order, because these models do not separate the three reading frames, i.e. the three different nucleotide positions in the codon.

It is possible to make a high order inhomogeneous Markov chain (discussed in the next section) for modelling the bases in three different reading frames, but since our goal is to score ORFs, we will do it differently. The sequences are transformed to sequences of codons. An arbitrary symbol is assigned to each of the 64 codons, and all genes and NORFs are translated to this alphabet (yielding sequences of one-third the length of the nucleotide sequences). Notice that this transformation is slightly different from the one above for transforming an *n*th order model into a first order one, because the triplets are non-overlapping.

A 64-state first order Markov chain was estimated from the translated sequences and tested on the genes in the test set and the NORFs in exactly the same way as the models above. The result is shown in Figure 3.12. Although the separation is not perfect, we see that it is much better than for the other model. Notice that the distribution we compare to in the log-odds score now is a uniform distribution over codons. The grey peak is centred around 0, indicating that the Markov chain has found a signal that is special to coding regions, and that codon usage is essentially random in the average NORF, and that a significant fraction of the NORFs scoring highly represent real genes that are not labelled as such in our data. It is likely that most of the ORFs scoring above 0.3–0.35 bits in this plot are overlapping with real genes. The NORF histogram uses a smaller bin size (as in Figure 3.11), and if the same bin size was used, the NORF histogram would be about five times higher.

Change begin

If the log-odds is not normalised by sequence length the discrimination improves significantly, because real genes tend to be longer than NORFs, see Figure 3.12.

Change end

Exercises

3.10 Calculate the number of parameters in the above codon model. The dataset contains on the order of 300 000 codons. Would it be feasible to estimate a second order Markov chain from this dataset?

3.11 How can the above gene model be improved?

Inhomogeneous Markov chains

As we saw above, a successful Markov model of genes needs to model the codon statistics. This can also be done without translating to another alphabet. It is well known that in genes the three codon positions have quite different statistics, and therefore it is natural to use three different Markov chains to model coding regions. The three models are numbered 1 to 3 according to the position in the

‘would be more natural’ changed to ‘is natural’.

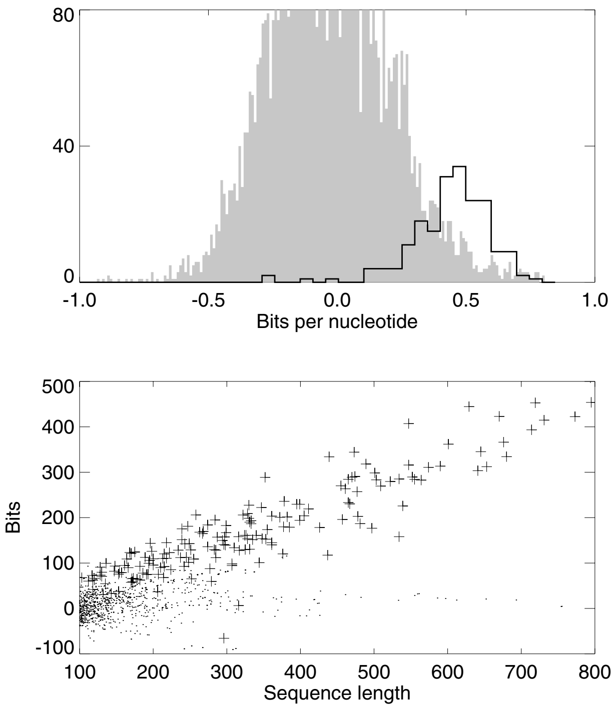


Figure 3.12 *The top plot shows the histograms of NORFs and genes for the Markov chain of codons (cf. Figure 3.11). Below, the log-odds is shown as a function of length for genes (+) and NORFs (·).*

codon. Assuming that x_1 is in codon position 3, the probability of $x_2, x_3, \ldots$ would then be

$$a^1_{x_1x_2} a^2_{x_2x_3} a^3_{x_3x_4} a^1_{x_4x_5} a^2_{x_5x_6} \cdots$$

where the parameters for model k are called a^k. This is called an *inhomogeneous Markov chain*. Here we assumed the chain was first order, but it is of course possible to extend it to order n. The estimation of the parameters is a straightforward extension of the estimation of the homogeneous models described in Section 3.1: for a second order inhomogeneous Markov chain as above the parameters of model 1 are estimated by counting the triplets with the last base in codon position 1, and similarly for model 2 and 3.

Change begin

Change end

Inhomogeneous Markov chains are used extensively in the GENEMARK genefinding program [Borodovsky & McIninch 1993], which is currently the most widely used method for prokaryotic genefinding. Inhomogeneous models of order up to five of coding regions have been combined with homogeneous models of the non-coding regions to localise genes in a number of different bacterial genomes.

'whole bacterial genome sequences' changed to 'different bacterial genomes'.

The first order model described above can also be constructed as an HMM, with the number of states equal to three times the length of the alphabet (a total of 12 for DNA). Higher order models can be made by adding many additional states to the HMM. However, it is also possible to have nth order Markov emission probabilities in the states of an HMM, in which the emission probabilities are conditioned on the n previous characters, so the emission probabilities (3.5) become

Change begin

$$e_k(b|b_1,\ldots,b_n) = P(x_i|\pi_i = k, x_{i-1} = b_1,\ldots,x_{i-n} = b_n).$$

All the algorithms derived for standard HMMs can be used with only obvious alterations for models with these emissions. Such models are also being used for genefinding [Krogh 1998].

Change end

Exercise

3.12 Draw the HMM that corresponds to the first order inhomogeneous Markov chain given above.

3.6 Numerical stability of HMM algorithms

Section heading changed. Also headings below.

Even on modern floating point processors we will run into numerical problems when multiplying many probabilities in the Viterbi, forward, or backward algorithms. For DNA for instance, we might want to model genomic sequences of 100 000 bases or more. Assuming that the product of one emission and one transition probability is typically 0.1, the probability of the Viterbi path would then be of the order of $10^{-100\,000}$. Most computers would behave badly with such numbers: either an underflow error would occur and the program would crash; or, worse, the program would keep running and produce arbitrary wrong numbers. There are two different ways of dealing with this problem.

The log transformation

For the Viterbi algorithm we should always use the logarithm of all probabilities. Since the log of a product is the sum of the logs, all the products are turned into sums. Assuming the logarithm base 10, the log of the above probability of $10^{-100\,000}$ is just $-100\,000$. Thus, the underflow problem is essentially solved. Additionally, the sum operation is faster on some computers than the product, so on these computers the algorithm will also run faster.

We will put a tilde on all the model parameters after taking the log, so for example $\tilde{a}_{kl} = \log a_{kl}$. Then the recursion relation for the Viterbi algorithm (3.8) becomes

$$V_l(i+1) = \tilde{e}_l(x_{i+1}) + \max_k(V_k(i) + \tilde{a}_{kl}),$$

where we use V for the logarithm of v. The base of the logarithm is not important as long as it is larger than 1 (such as 2, e, and 10).

It is more efficient to take the log of all the model parameters before running the Viterbi algorithm, to avoid calling the logarithm function repeatedly during the dynamic programming iteration.

'forward–backward algorithm' changed to 'the forward and backward algorithms

For the forward and backward algorithms there is a problem with the log transformation: the logarithm of a sum of probabilities cannot be calculated from the logs of the probabilities without using exponentiation and log functions, which are computationally expensive. However, the situation is not in practice so bad. Assume you want to calculate $\tilde{r} = \log(p+q)$ from the log of the probabilities, $\tilde{p} = \log p$ and $\tilde{q} = \log q$. The direct way is to do $\tilde{r} = \log(\exp(\tilde{p}) + \exp(\tilde{q}))$. By pulling out $\tilde{p}$, one can write this as

$$\tilde{r} = \tilde{p} + \log(1 + \exp(\tilde{q} - \tilde{p})).$$

It is possible to approximate the function $\log(1+\exp(x))$ by interpolation from a table. For a reasonable level of accuracy, the table can actually be quite small, assuming we always pull out the largest of $\tilde{p}$ and $\tilde{q}$, because $\exp(\tilde{q} - \tilde{p})$ rapidly approaches zero for large $(\tilde{p} - \tilde{q})$.

Scaling of probabilities

An alternative to using the log transformation is to rescale the f and b variables, so they stay within a manageable numerical interval [Rabiner 1989]. For each i define a scaling variable s_i, and define new f variables

$$\tilde{f}_l(i) = \frac{f_l(i)}{\prod_{j=1}^{i} s_j}. \tag{3.26}$$

From this it is easy to see that

$$\tilde{f}_l(i+1) = \frac{1}{s_{i+1}} e_l(x_{i+1}) \sum_k \tilde{f}_k(i) a_{kl},$$

so the forward recursion (3.11) is only changed slightly. This will work however we define s_i, but a convenient choice is one that makes $\sum_l \tilde{f}_l(i) = 1$, which means that

$$s_{i+1} = \sum_l e_l(x_{i+1}) \sum_k \tilde{f}_k(i) a_{kl}.$$

The b variables have to be scaled with the same numbers, so the recursion step in (3.3) becomes

$$\tilde{b}_k(i) = \frac{1}{s_i} \sum_l a_{kl} \tilde{b}_l(i+1) e_l(x_{i+1})$$

This scaling method normally works well, but in models with many silent states, such as the one we describe in Chapter 5, underflow errors can still occur.

Exercises

3.13 Use (3.26) to prove that $P(x) = \prod_{j=1}^{L} s_j$ with the above choice of s_i. It is of course wiser to calculate $\log P(x) = \sum_j \log s_j$.

3.14 Use the result of the previous exercise to show that the equation (3.20) actually simplifies when using the scaled f and b variables. Also, derive the result (3.21) for the scaled variables.

3.7 Further reading

Added one line with references.

More basic introductions to HMMs include Rabiner & Juang [1986] and Krogh [1998].

Change begin

Change end

change of wording and Burge ref added.

Some early applications of HMM-like models to sequence analysis was done by Borodovsky *et al.* [1986a; 1986b; 1986c] who used inhomogeneous Markov chains as described on p. 76. This later led to the GENEMARK genefinder program [Borodovsky & McIninch 1993]. Cardon & Stormo [1992] introduced an expectation maximisation (EM) method, which has many similarities with an HMM, for modelling protein binding motifs. Later applications of HMMs to genefinding include Krogh, Mian & Haussler [1994], Henderson, Salzberg & Fasman [1997], and Krogh [1997a,1997b,1998] as well as systems combining neural networks and HMMs [Stormo & Haussler 1994; Kulp *et al.* 1996; Reese *et al.* 1997; Burge & Karlin 1997]. Such hybrid systems are also becoming quite popular for other applications; see for instance Bengio *et al.* [1992], Frasconi & Bengio [1994], Renals *et al.* [1994], Baldi & Chauvin [1995], and Riis & Krogh [1997].

Change begin

Change end

Change begin

Change end

Churchill [1989] used HMMs for modelling compositional differences between DNA from mitochondria and from the human X chromosome and bacteriophage lambda, and later for studying the compositional structure of genomes [Churchill 1992]. Other applications include a three-state HMM for prediction of protein secondary structure [Asai, Hayamizu & Handa 1993], a HMM with ten states in a ring for modelling an oscillatory pattern in nucleosomes [Baldi *et al.* 1996], detection of short protein coding regions and analysis of translation initiation sites in cyanobacteria [Yada & Hirosawa 1996; Yada, Sazuka & Hirosawa 1997], characterization of prokaryotic and eukaryotic promoters [Pedersen *et al.* 1996], and recognition of branch points [Tolstrup, Rouzé & Brunak 1997]. Several other applications of HMMs will be discussed in the context of profile HMMs in Chapters 5 and 6.